CMPE333 Project

Baseline

I used a basic decision tree as a baseline for performance. As this was just a baseline I simply ignored non-numeric columns such as *mn_sat* and missing values. I had some trouble getting this to work until I found that the *dec* column mirrored the *match* column, then I excluded that too. This gave an accuracy of 88%, which still seemed very good. I found and removed the *dec_o* field, and the accuracy was lowered to 78%.

From there I looked at the decision tree, and saw that it was using some arbitrary fields such as *iid* early in its decision process. Removing those increased the performance to 80%.

match \ Pr...	0	1
0	2173	237
1	347	176

Correct classified: 2,349 Wrong classified: 584

Accuracy: 80.089 % Error: 19.911 %

Cohen's kappa (κ) 0.26

Python data manipulation

Once I established my baseline, I created a python script to do some data cleanup and generate more features. The first thing I did was eliminate the commas from the zipcode, income, mn_sat and tuition columns so that Knime could read them properly.

I also accumulated a list of people, then calculated the distance between a pair's zipcodes, and appended columns with the differences in each interest between the two people. This script created a (very) small increase in performance, and adding in some fields such as *income* actually decreased it slightly.

Knime data manipulation and exploration

I then used Knime to do a bit more data manipulation. This included playing with which columns to include, and normalizing the preference columns to try to handle the different points allocation schemes for those columns. I was unable to get any performance increase with these changes.

I then decided to explore the data itself a bit more. I found that most participants were in their twenties, with a mean of 26. I then looked at how people valued the different attributes in a partner, with the results shown below:

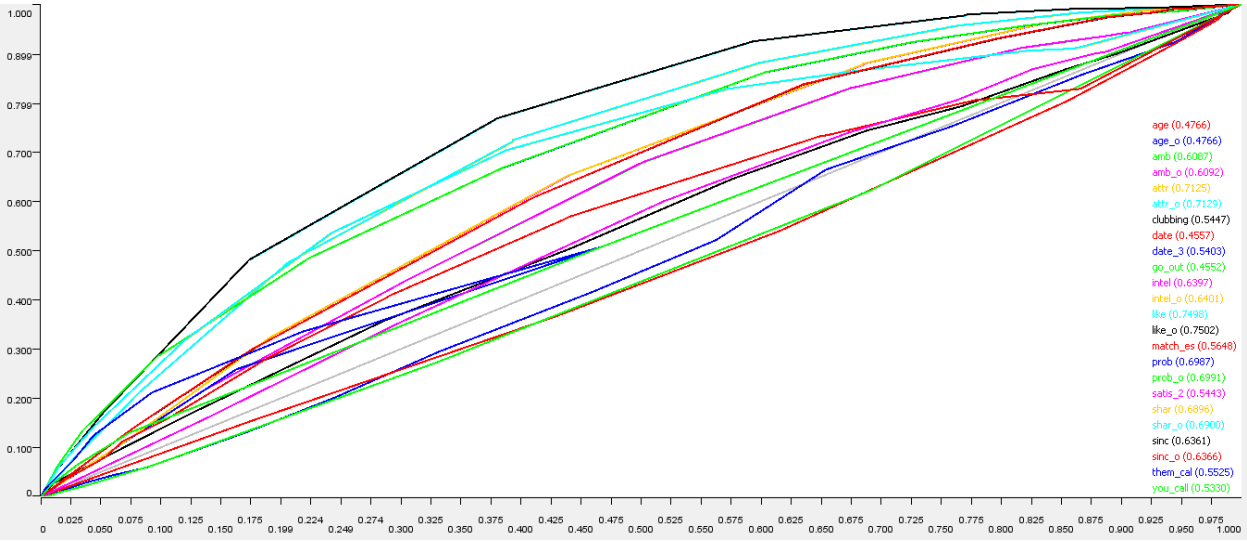
Attribute	Average Importance
Attractiveness	22.5
Ambition	10.7
Fun	17.5
Intelligent	20.3
Shared Interests	11.8
Sincerity	17.4

This makes it easy to see what people value – attractiveness and intelligence are high, ambition and shared interests are low. This can be explained by looking at the Goal field – most people are looking for a fun night or to meet new people, and very few are looking for serious relationships. Ambition and interests might be more serious if you're looking to get married, but very few of the participants are.

I found some interesting discrepancies in peoples' ratings of the importance of attractiveness. People valued attractiveness highly – averaging 22 points. However, regardless of gender people thought the other sex valued it higher at 26 points. When it came to perception of attractiveness, people rated themselves on average 7, while they rated their partners lower at 6.2. I did not separate this by gender, though I might do that later in the project.

ROC curve

The next step was to generate a ROC curve. The first one with all columns was a mess, so I took a couple rounds of eliminating the least beneficial columns. I ended up with the following most important fields: like_o, like, attr, attr_o, prob and prob_o. These all make quite a bit of sense, as most of them relate directly to how much the people liked each other.



Split Normalization

The data is not uniform – for several of the questions, waves 6-9 were given different instructions than the other waves. To address this, I split the data based on the wave and normalized the two sets separately. This is not foolproof, as the scoring is completely different for the two groups, but it should help avoid the case where one person gave 100 points in one area and all the people who were rated out of 10 are normalized to very low values.

Decision Forest

In my (limited) experience, decision forests have been very effective for categorization. This was the case for the assignments earlier in this course, as well as for my work this summer on bus ridership predictions. This immediately gave a performance of 85%, better than previous methods.

Since the start of the project I have gotten a significantly higher-powered computer, so I figured I may as well have some fun and push it a bit. I ended up being disappointed, as increasing the number of models in the forest from 100 to 400 had only a small impact – 85.7 compared to 85.6 accuracy. Curious, I decreased the number of models to 10, and still got an accuracy of 84.5. I decided to plot the accuracy against the number of models – shown in Figure 1. You can see that the difference between 1 and 2 is huge, then for some reason 3 is bad, then it is a slow increase from there. There are small gains to be made between 10 and 100, then only fractions of a percent from 100 to 1000.

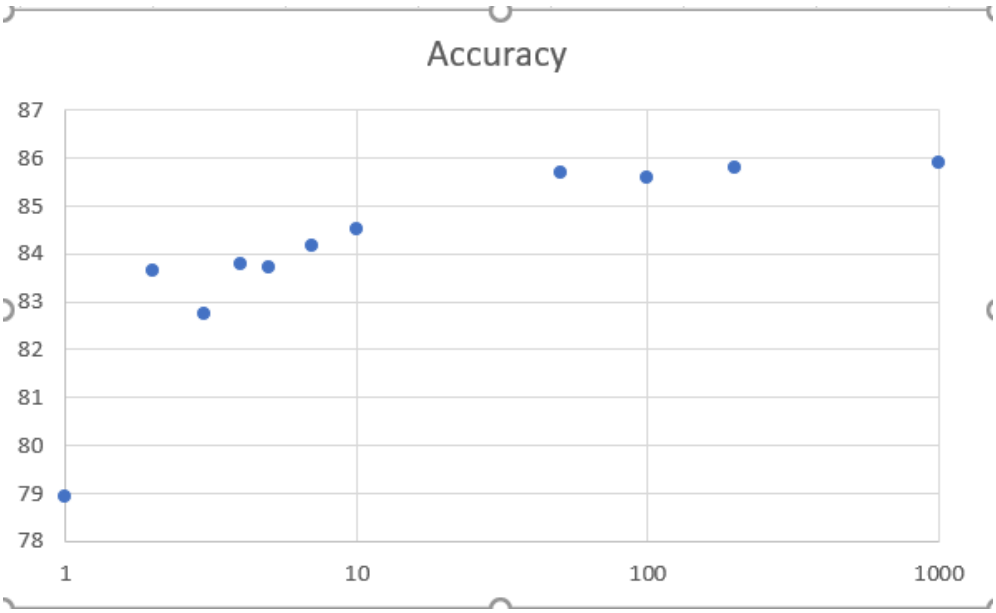


Figure 1 - Accuracy vs number of models in tree

I settled on 200 models as the gain beyond that point was so small, but it was still fairly quick. I then played with other parameters. The default split criterion (information gain ratio) turned out to be better than Information Gain and roughly tied with Gini index. I didn't touch the number of levels, as this is a minimum that I think is only used to improve resource usage.

Parallel Learners

At this point I think it is a good idea to just create a number of different learners and run them in parallel. This was I can see how different manipulations affect each model. So, I added many types of learners with default values to see how they did. The results are summarized in Table 1.

Table 1 - Default performance of many models

Learner Type	Accuracy	Comments
Naïve Bayes	82.8%	
Fuzzy Rule	81.95%	Slow to predict
Gradient Boosted Trees Regression	83.6%	
Gradient Boosted Trees	86.8%	
Random Forest Regression (RFR)	4.7%	Gives values between 0 and 1 rather than

		categories. Mean squared error is 0.122.
Decision Tree applied to the RFR predictions	84.8%	Shows that the RFR is decent, just the scorer wasn't showing the whole story
Tree Ensemble	86.5%	
Tree Ensemble Regression	5.2%	Same as the RFR. Also same mean squared error of 0.122.
RPROP MLP	85.5%	
SVM	85.9%	Slowish
PNN	83.6%	Veeery slow, I went to get groceries while it ran. Will not try again.

Based on these results, we see that the Gradient Boosted Trees and Tree ensemble have very good performance out of the box, even beating the Decision Forest I tuned. It might be worth focusing on them going forward.

Pairing Data

On the suggestion of a friend, I decided to try pairing the data. Each row corresponds to an interaction between two people – identified by the IID and the PID. For each of these rows, I found the other row for the same interaction, which had the IID and the PID switched, and concatenated it onto the original row with the prefix 'partner_' on each column name. This was done in python, and took quite a while with all the querying.

This technique initially gave a 100% success rate with most models, which told me I need to remove some columns. I looked at the Decision Tree model to see which columns it was using, and removed *partner_match* and *partner_dec_o*.

Once I cleaned up the obvious columns I reran all the models with the new paired data. The results, which are shown in Table 2, were mixed. RFR, MLP, and SVM did worse or the same, and most other models had modest improvements. The best model is now the Gradient Boosted Trees model, with an accuracy of 87.8%.

Table 2 - Results of pairing data

Learner Type	Previous Accuracy	Paired Accuracy
Naïve Bayes	82.8%	83.9%
Fuzzy Rule	81.95%	84.6%
Gradient Boosted Trees Regression	83.6%	84.1%
Gradient Boosted Trees	86.8%	87.8%
Decision Tree applied to the RFR predictions	84.8%	77.5%
Tree Ensemble	86.5%	87.3%
RPROP MLP	85.5%	85.4%
SVM	85.9%	85.8%

Missing Values

So far I hadn't done much about missing values. For the models that could no handle missing values I filled in fixed values of 0, but that doesn't add any value. I decided to try inserting the average, but this actually had a slight negative effect. Those averages probably represented false tracks, making it hard for the models to learn accurately.

I then tried splitting the data by gender and adding the averages by gender. This worked better than flat averages, but still less than leaving the empty values. This is probably because my most performant models can accept missing values, so adding an average actually removes possibly useful information. The models that cannot handle missing values, such as the SVM, performed about the same as with the zeroes.

Best model tuning

At this point I had tried various data manipulations and models. My best models gave accuracies over 87%, which seems pretty good considering how messy the data is and how it is based on naturally fickle human behaviour. I decided to narrow my focus to the best model – the gradient boosted tree (GBT). I tweaked the configuration to maximize the performance. For example, increasing the number of models in the GBT improved the accuracy to more than 88%. However, this slowed it down considerably, so I tested all other parameters with a much smaller number of models. Sadly, the other tweaks had minimal effect and the GBT maxed out at 88.3 accuracy.

Best Model Tuning

I stuck with the GBT as my final model. It is somewhat slow to train, but fast to predict and that's what matters in production. The results of this model are shown below:

match \ Prediction (match)	0	1
0	1681	51
1	195	168

Correct classified: 1,849 Wrong classified: 246

Accuracy: 88.258 % Error: 11.742 %

Cohen's kappa (κ) 0.514

As you can see, it has an accuracy of 88.258%. I had slightly higher runs around 88.33%, but this one was stable. As you can see, the false results are not evenly balance – about 80% of them are false negatives. That means that the model rarely predicts *yes* when the person says no, but it can miss some yesses. That isn't too surprising, for a couple reasons. One has to do with psychology. It seems to me that people usually have pretty obvious reasons to say *no* to a date. However, sometimes people might give a *yes* on a whim, or because they felt something they can't quite identify. This doesn't happen as often for *no*, but it would be very hard for a model to figure out the reasoning the person themselves can't explain. Thus, the model misses these cases and gives false negatives.

The other, probably more significant reason is that the dataset is biased towards *no*. More than 80% of the responses are *no*, so the model does well if it over-predicts *no*. I decided to try upsampling the data to see whether that would improve it or affect the ratio of error types.

Upsampling

I used SMOTE to increase the number of positives, but set it up wrong so I ended up with a TON of samples:

Row ID	match_d	Count (...
Row0	0	20994
Row1	1	4140

That wasn't what I wanted, and it didn't change the ratio, but it had a fantastic accuracy level:

match \ Pr...	0	1
0	5167	43
1	183	891

Correct classified: 6,058 Wrong classified: 226

Accuracy: 96.404 % Error: 3.596 %

Cohen's kappa (κ) 0.866

Cool! I decided to give SMOTE one more try, specifying to upsample the minority classes, which was what I wanted. SMOTE takes forever, so this was my last chance. SMOTE created more entries with *yesses*, resulting in this:

Row ID	D match_d	I Count (...
Row0	0	6998
Row1	1	6998

This also had very good results:

match \ Pr...	0	1
0	1605	96
1	87	1711

Correct classified: 3,316 Wrong classified: 183

Accuracy: 94.77 % Error: 5.23 %

Cohen's kappa (κ) 0.895

It is worth noting that while the first round of SMOTE, which did not change the ratios, led to a model that still overpredicted *no*. Meanwhile, the second round which balanced the responses had a fairly even split between false positives and false negatives. This confirms my theory that the skewed errors were due to the ratio of responses. These are very good results, though they most likely come from the fact that there is now more data to train and test on. Further manipulations might involve changing SMOTE to keep the total amount of data the same, or only SMOTEing the training data. However, I am very happy with these results and the deadline is looming so this will have to do.

Conclusion

This is a challenging data set due to a number of factors – abundant missing data, varying scoring methods, many basically useless attributes, and the fickleness of the human heart to name a few. With some data cleanup and preparation, various models were able to provide more than 80% accuracy. The best model was a Boosted Gradient Trees model, which provided accuracy around 88%. This was considerably improved with the use of SMOTE to balance out the distribution of the class attribute, *match*. This offered considerable improved accuracy approaching 95%, but at least some of this improvement can likely be attributed to the increased quantity of training data provided by SMOTE.