Capstone Project

Vince Favilla

September 12, 2017

I. Definition

Project Overview

For this project, I am analyzing data from speed dating events and creating a model to predict whether two people will be interested in seeing each other again.

As a psychology professor, I'm obviously fascinated by human behavior. I became interested in machine learning and data science because I saw it as an innovative new way to understand why we make the decisions we do. And, when it comes to dating and relationships, humans are wildly unpredictable. We routinely make romantic decisions that go against our own stated preferences -- see, for instance, the work of Eastwick and Finkel (2008).

On the other hand, some general principles do in fact predict romantic relationships. Dr. David Buss is a pioneer in the field of "mate selection" -- the psychological term for choosing a partner. One of the most robust findings in his research is that of **homogamy**: people tend to pair with others that are similar to themselves in a variety of dimensions: personality, religion, political opinions, education, income, race, and so on.

I'm using the Speed Dating Experiment dataset available on Kaggle. This is from a study performed by professors Ray Fisman and Sheena Iyengar for their paper Gender Differences in Mate Selection: Evidence From a Speed Dating Experiment. The data was gathered from several speed dating events conducted from 2002 to 2004 and contains 194

features in addition to the “match” target variable. Each row contains a record of one speed dating encounter: the participant’s ID, their partner’s ID, and information about the participant.

Some of the more noteworthy features include:

- Hometown
- Socioeconomic status/income (as predicted by the zip code they’re from)
- Self-rated personality traits of ambitiousness, fun-ness, & sincerity
- Importance of those same traits in a partner
- Self-rated attractiveness
- Importance of attractiveness in a partner
- Planned career path (these are college students, after all)

There are about 8,400 observations in the dataset, so we have plenty of data to learn from.

Problem Statement

Given the demographic data and personal preferences of a man and woman, can we predict whether they’ll be interested in one another romantically?[1]

This problem is easily quantified:

- Our **target variable** is binary: “1” if both individuals would like to meet again, and “0” if one or both people decline.
- For **demographic** data, we can use categorical variables for information such as hometown, career, or race. We can also use U.S. census data to predict socioeconomic status based on one’s zip code.
- For **preferences, personality and hobbies**, people can give numerical answers to various statements. For instance: “How

[1] I’d happily study gay and lesbian couples as well, but our dataset is for a heterosexual speed dating event.

> ambitious are you, from 1-10?" or "How much do you like movies, from 1-10?"

This is one of my favorite aspects of psychology. Researchers have spent the last century figuring out how to quantify our thoughts and behavior. We could measure an arbitrarily large number of features just by following the template above.

I solved this problem by using my domain knowledge to augment the dataset with new features, and then tested a variety of algorithms before settling on an ensemble-based learner (XGBoost) combined with feature selection (SelectPercentile).

Metrics

I evaluated my model using the f-beta score with a beta of 1.5. F-beta is a versatile scoring metric that requires both precision and recall to obtain a high score.

I chose a beta of 1.5 because I want to slightly favor recall over precision. This means that I want my model to err on the side of false positives rather than false negatives (some of its predicted matches will be incorrect).

Let's put ourselves in the shoes of a prospective dater: When it comes to dating, the cost of a false negative could be missing out on a partner who would be a great match for you. Meanwhile, the cost of a false positive is only a "wasted" evening -- and even then, many daters don't consider a pleasant conversation with someone to be a waste of time. So that's my rationale for selecting a beta of 1.5.

This is how we'll compute it:

$$F_\beta = (1 + \beta^2) \frac{\text{Precision} * \text{Recall}}{(\beta^2 \cdot \text{Precision}) + \text{Recall}}$$

(credit: Udacity)

Where β = 1.5.

II. Analysis

Data Exploration

The data is from 21 separate events, and a few of these events recorded data slightly the differently. I'll explain this in a moment, but let's take a look at the data first.

The original dataset has 8378 observations and 195 features. After removing some unnecessary rows and columns (described below), we end up with 5761 observations and 179 features. I split this data into 80% training and 20% testing.

The original features have the following data types:

- float64: 174
- int64: 13
- object: 8

It's impractical to list the statistics for all 194 features, so I'll just share the stats for the more noteworthy ones and features that were most important in my final model.

Match distribution

- 0: 0.835
- 1: 0.165

Income

- Mean: 44887.61
- SD: 17206.92
- Max value: 109031.0
- Min value: 8607.0

Age

- Mean: 26.36
- SD: 3.57
- Max value: 55.0
- Min value: 18.0

Gender

- man: 277
- woman: 274

Partner's stated importance of attractiveness

pf_o_att

- Mean: 22.50
- SD: 12.57
- Max value: 100.0
- Min value: 0.0

Partner's stated importance of intelligence

pf_o_int

- Mean: 20.27
- SD: 6.78
- Max value: 50.0
- Min value: 0.0

Partner's stated importance of fun-ness

pf_o_fun

- Mean: 17.46
- SD: 6.09
- Max value: 50.0
- Min value: 0.0

Partner's stated importance of sincerity

pf_o_sin

- Mean: 17.40
- SD: 7.04
- Max value: 60.0
- Min value: 0.0

Partner's stated importance of shared interests

pf_o_sha

- Mean: 11.85
- SD: 6.36
- Max value: 30.0
- Min value: 0.0

Partner's stated importance of ambition

pf_o_amb

- Mean: 10.69
- SD: 6.13
- Max value: 53.0
- Min value: 0.0

Importance of race

imprace

- Mean: 3.78
- SD: 2.85
- Max value: 10.0
- Min value: 0.0

Importance of religion

imprelig

- Mean: 3.65
- SD: 2.81
- Max value: 10.0
- Min value: 1.0

Notice how the “stated importance” features have means around 15-20. Daters were given 100 points to distribute to these features as they saw fit.

In few of the events, however, daters were simply asked to rate these features from 1-10. I tried to normalize this scoring across all of the events, but this was rather ineffective. Instead, it was better to simply remove the 6 events where people used the 10-point scale.

In preparing the data, I dummified the categorical variables, and then removed dummified features where a “1” only appeared once. I didn’t need to remove any outliers.

There were a lot of missing values in this data set. I used a K-nearest neighbors regression algorithm to predict them, but also used the median if a cross-validation showed that my regression didn’t work. This strategy worked the best, compared to a) replacing missing values with the mean, and b) replacing missing values with -100 and letting a tree-based algorithm deal with them later.

Finally, I removed some features that were a little too similar to my target variable. I selected these by looking for correlations above 0.25 (this value was obtained through trial and error). They were features such as:

- “Overall, how much do you like this person?”
- “How probable do you think it is that this person will say 'yes' for you?”
- “How likely are you to call this person?”

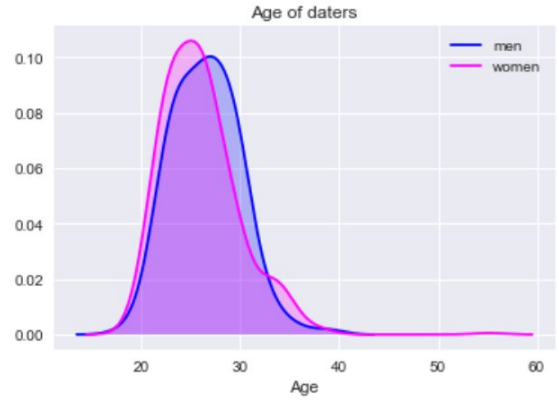
- “How likely do you think this person is to call you?

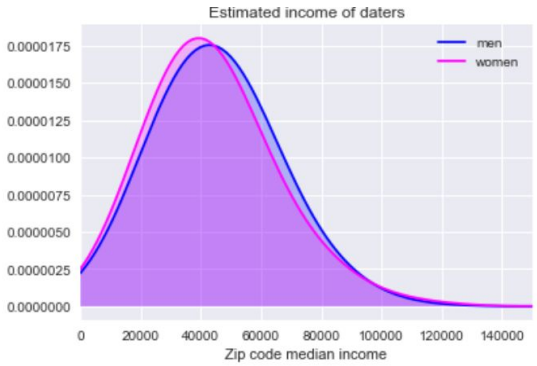
Naturally, questions such as these would predict matches a little too easily and would not mimic a real-world scenario, such as on a dating app.

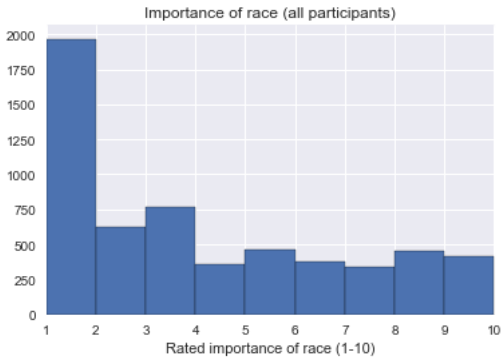
Exploratory Visualization

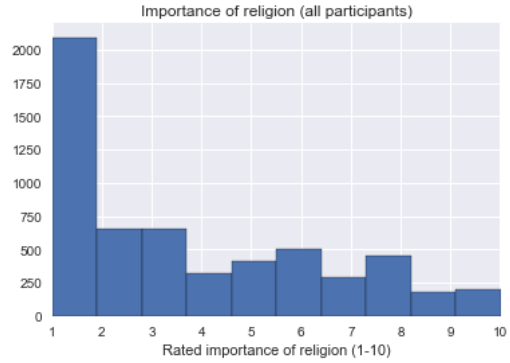
I tried plotting various features against the target variable, but none of the visualizations were especially interesting or helpful.

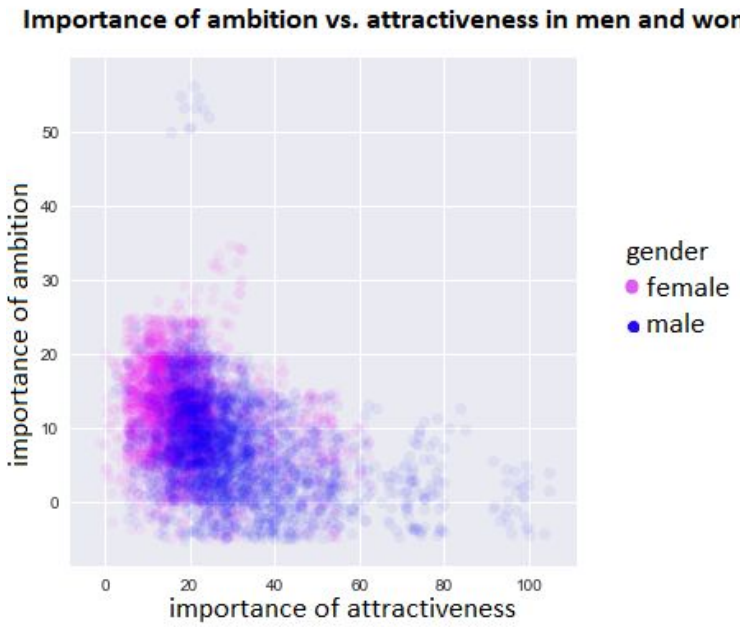
Instead, I’ll share a few visualizations that illustrate the demographics of the participants.











Some thoughts: I really like that last graph. Consistent with my intuition, women value ambition more and men value attractiveness *much* more. However, there are a handful of men who place extremely high importance on ambition -- they must be looking to be part of a power couple!

I *didn't* like the importance of race and religion graphs. With the distribution we see, I suspect it might be better as a 3, 4, or 5-point scale.

Algorithms and Techniques

I tested a variety of algorithms, and used a grid search to find their best settings. Here are the ones I used, along with brief explanations of how they work:

`KNeighborsClassifier(n_neighbors=1)`

K-nearest neighbors (KNN) compares data in the testing set to data in the training set. It identifies the *k* data points in the training data that

are most similar to each observation in the testing data, and then "polls" these matches to see what they think the label should be.

With k=1, the algorithm finds the single closest match in the training data, and then applies that match's label to the test observation.

`GaussianNB()`

Naive Bayes treats each feature as an independent observation that increases or decreases confidence in the predicted label. I really like Wikipedia's example:

> "For example, a fruit may be considered to be an apple if it is red, round, and about 10 cm in diameter. A naive Bayes classifier considers each of these features to contribute independently to the probability that this fruit is an apple, regardless of any possible correlations between the color, roundness, and diameter features." (Source)

So, under the hood, GaussianNB is doing something like this:

- Prior probability that it's an apple:
  0.20
- Probability that it's an apple given it's red:
  0.70 (it might be a cherry!)
- Probability that it's an apple given it's not spherical:
  0.80 (it still might be a strawberry!)
- Probability that it's an apple given it's 10cm in diameter:
  0.98

Replace "red and spherical" with "both like yoga and have the same income," and you have the idea of Bayesian matchmaking!

`LogisticRegression(C=100, penalty='l1')`
Logistic regression weighs each relevant feature with a learned coefficient before adding up these values (and then passing them

through a sigmoid function) to predict the probability of a positive observation.

Mathematically, I consider regression to be one of the simplest algorithms. It's the first one I learned, and I still occasionally find it outperforming its more complicated cousins.

`DecisionTreeRegressor(max_depth=None, min_samples_leaf=1)`

Decision trees are essentially playing "20 Questions" with the data. It splits the data by finding the optimal yes/no questions it can ask. For instance:

- Are they estimated to have incomes within $10,000 of each other?
- Are their ages no more than 3 years apart?
- Do they both rate the importance of sincerity above a 15?

If the answer to all 3 questions is "yes," then perhaps we'd predict a match. Otherwise, we would not. (This is just an example.)

As a side note, these optimal settings for a decision tree were really surprising to me. They have "overfitting" written all over them! I suppose this illustrates the importance of cross-validation. Sometimes your hyperparameters surprise you.

`RandomForestClassifier(max_depth=None, min_samples_leaf=2, n_estimators=10)`

If a decision tree is like playing 20 Questions with someone, Random Forest is like playing "5 Questions" with a few different people, each with their own area of expertise. At the end, each player (i.e., estimator) votes on whether the observation is a 1 or 0.

Again, these hyperparameters are a little surprising. Rather than 5 or 20 Questions, GridSearchCV suggests I play "Infinite Questions" (max_depth) with 10 people!

`XGBClassifier(base_score=0.44, max_depth=9, n_estimators=100)`

Gradient boosting (as implemented by XGBoost) is another ensemble method. It adds new learners until it can no longer improve its predictions. It trains on the residuals (errors) of previous models in order to give added weight to data points the algorithm can't currently predict. It uses gradient descent to minimize its prediction errors.

Benchmark

I used sklearn's DummyClassifier to predict a "1" for each record in the dataset. If we were measuring accuracy, this would be a terrible strategy because most pairs didn't experience a love connection. But because we'll be weighting our metric towards recall, this will provides an appropriate and useful benchmark.

The DummyClassifier obtained an f-beta score of approximately **0.396**.

III. Methodology

Data Preprocessing

Below, I describe all of the pre-processing steps I took. It may be more helpful to view them (along with my comments) in the Jupyter notebook itself.

1. As described earlier, I removed the events where participants used a 10-point scale rather than distributing 100 points. This eliminated 6 of the 21 events in the dataset.
2. As also described, I removed features that were a little too similar to our "match" target variable. This included 16 of the 195 features present.
3. Also removed was the "field" feature, which was a categorical variable describing someone's career plans. This was redundant because there is already another feature that encodes this information numerically.

4. There were 4 features that encoded numeric data as a string. I converted these features back to floats and replaced missing values with -100. This strategy worked better than simply entering the mean or median value.
5. For remaining string-based columns, I converted the text to lowercase so I could process the data more reliably.
6. For missing data, I used a k-neighbors regressor to predict the missing values. Naturally, I did not use my target variable in the regression because that would be looking into the future and would contaminate the data.
   I chose to use a k-neighbors regressor because it's computationally cheap and there was a lot of data to process. When a cross-validation suggested that the regression was ineffective, I instead used the feature's median value.
7. For the features I dummified, I removed features where only one person had a "1" value. A cross-validation showed that this improved the performance of the model, while also reducing dimensionality.

Feature engineering

1. I examined people's responses to the three features "from," "undergra(d)," and "career," and combined some of the more common answers into a single variable -- for example, "from_china." I would first set the default value of these features to zero, and then overwrite it with a 1 when a response matched one of the strings listed ('china', 'beijing', 'shanghai', etc.).
2. I then dummified the remaining data using `pd.get_dummies()`.
3. The function `get_partner_data()` does what the name suggests. Since each row contains a participant ID and their partner's ID, this function concatenates the partner's demographic data.
4. I added a (log) income difference feature based on partner data. When later analyzing feature importances, this turned out to be the most important feature in the entire dataset.

5. I also added an age difference feature.
6. I used a few different methods to compare one's expectations (or, more accurately, priorities) to the person they were meeting. I calculated:
    a. The difference between the partner's self-rated qualities to the person's stated importance of those qualities.
    b. Another version of "a," except using data that participants entered halfway through an event. For instance, people were asked to re-evaluate how attractive or sincere they thought they were. I figured there was a possibility that this updated data would vary significantly from the original ratings, so I included it as well.
    c. I binned people's self-ratings and priorities into buckets of "below average" and "above average." So, for instance, if someone placed an above-average importance on attractiveness, and a partner rated themselves more attractive than average, I encoded the met expectations as a "1." Otherwise, the feature would score a "0."
    d. I tried one additional technique where I compared the importance of ambitiousness to the median income in one's childhood zip code. I believed this would be a way to compare socioeconomic status.
7. Using the partner data, if a participant and their partner both went to the same school, are from the same place, or have similar career ambitions, I assigned a "1" for an additional feature, otherwise I assigned a "0."
8. For the hobbies recorded in the dataset, I did something similar. If, for instance, both people liked yoga, the `yoga_in_common` feature got a "1."
9. I broke down the 5 digits of one's childhood zip code. Originally, I planned to compare zip codes to see how far apart people grew up. It would be something along the lines of:
    - 5 digits in common: Same neighborhood

- 4 digits in common: Same city
- 3 digits in common: Same state

and so on. But what I found was that comparing zip codes like this actually *reduced* the model's accuracy, while simply breaking down zip code components and adding the partner's data improved it.

As such, I created new features for the first 4 digits of the zip code, first 3 digits, and first 2. I also concatenated this information from one's conversation partner.

Feature selection & transformation

I tested two methods of dimensionality reduction: principal component analysis and SelectPercentile. The latter was more effective.

I tried processing the data with a min-max scaler, but this did not improve the model's performance. So I kept the data in its original form.

Algorithm selection

After cleaning the data and engineering features, I tested the algorithms described earlier. I eventually settled on the XGBClassifier, because its initial cross validation looked promising, and XGBoost has a reputation for consistently performing well in a variety of contexts.

I then tested two methods of dimensionality reduction: principal component analysis and SelectPercentile. The latter, which only uses the most important features to make predictions, was significantly more effective and tended to select about 50% of the available features.

I combined these two steps into a pipeline, but I needed to take one final step to improve my model's performance. I created a custom

classifier in order to change the threshold at which it made positive predictions. This was done with sklearn's Base Estimator and Classifier Mix-in. My f-beta score improved when I lowered the threshold from predicting a "1" from 0.50 to about 0.48.

My final model, roughly speaking, was then composed of custom classifier wrapped around a **SelectPercentile** feature selector, and an **XGBClassifier** with 400 estimators.

Implementation

Most of the details of my implementation have already been documented, but I'll summarize the process here.

To evaluate my model's performance, I used f-beta score with a beta value of 1.5. To combine this with sklearn's grid search and cross-validation features, I used the following scorer:

```
from sklearn.metrics import make_scorer, fbeta_score

scorer = make_scorer(fbeta_score, beta=1.5)
```

I used a combination of `train_test_split`, `GridSearchCV`, and `cross_val_score` to test my model. I split my data in an 80% training set and a 20% testing set, and then used a grid search on my training data (typically with 4 folds) to find the optimal model.

The final evaluation of my model included **a)** evaluating the predictions of my test data, and then **b)** testing the model against *all* my data using `cross_val_score` (again with 4 folds).

Refinement

The only problem I ran into was described earlier: it turns out that using the 0.50 pred_proba threshold to predict a match was too high. To obtain a higher f-beta score, I needed to lower the threshold and predict matches a bit more liberally. This strategy was consistent with my original goal: to favor recall over precision.

I could have transformed these predictions manually, but instead I learned how to use sklearn's base estimator and classifier mix-ins to create a custom classifier. You'll see this in my notebook as the `MatchmakingClassifier`. Here is the "important" code inside the class that shows exactly how it works:

```
    def predict(self, X):

        pred = self.estimator.predict_proba(X)[:,1]

        pred = np.where(pred > np.percentile(pred,
                        self.threshold), 1, 0)

        return pred
```

When creating the class, you set a threshold at which it makes positive predictions. You can also pass along any algorithm that supports the predict_proba method.

Since predict_proba returns a probability distribution, we only use the positive prediction at index 1. Then we use `np.where` to convert any values above the threshold to a 1, while values below the threshold get a 0. Then we return the new predictions.

This class makes it very easy to continue using grid searches and cross val scores to evaluate the model. Despite having a new classifier, it continues to work with sklearn's existing features.

I used a fairly exhaustive grid search to optimize my original pipeline (SelectPercentile + XGBClassifier). I estimate that I ran the search for about 72 hours in total, and I'm fairly confident that I found the best settings. I abbreviate the parameters I used in the Jupyter Notebook so you can arrive at these settings without waiting for such a long time. So, although it might look like it's a narrow search, I did in fact look at many other possible settings:

- **SelectPercentile**: I looked from 5-100 in increments of 5. Then I tested increments of 2 near the optimal value it found (55).
- **N_estimators**: I tested 10, 50, 100, 200, 250, 300, 350, 400, 450, and 500. 400 was the optimal value.
- **Max_depth**: I tested values from 3 to 30. The optimal value seemed to vary a bit, but values of 7, 8, or 9 all seem to work pretty well.
- **Threshold**: This is the parameter in my custom classifier. I tested values from 40 to 60, and the optimal value is around 48.5.

To save a little time and simplify the workflow, I tested various thresholds *after* finalizing my pipeline's parameters. It's possible that an even better configuration exists, but I think it's unlikely given the time I put in, and the random re-tests I performed with the optimal settings. Most scoring differences were exceedingly small, anyway.

Challenges

The single greatest challenge was performing a proper grid search and having the patience to run it several times while continuously refining my settings. Luckily, the coding aspect of this was straightforward; it just required a lot of computation as well as keeping track of intermediate results.

Engineering new features was also fairly time consuming, but relatively straightforward. It took time to look through the text data and identify common themes (such as Ivy League schools or Chinese cities).

It took a little bit of creativity to structure this data and then concatenate the partner data, but I learned valuable skills in the process.

At some point I should also dig deeper into the original psychology study that used this data. I'm curious to see why the data is structured as it is; it wasn't always easy to parse mentally.

IV. Results

Model Evaluation and Validation

When it was all said and done, this was my final classifier:

```
MatchMakingClassifier(estimator=Pipeline(memory=None,
     steps=[

('reduce_dim', SelectPercentile(percentile=55,
         score_func=<function f_classif at 0x00000179C4F541E0>)),

('classify', XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
       colsample_bytree=1, gamma=0, learning_rate=0.1, max_delta_step=0,
       max_depth=7, min_child_weight=1, missing=None, n_estimators=400,
       n_jobs=1, nthread=1, objective='binary:logistic', random_state=0,
       reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=0, silent=True,
       subsample=1))

]),
       threshold=48.5)
```

On the testing data, this model obtains an f-beta score of 0.516, compared to 0.396 for the dummy classifier. This is a massive difference: it represents a 30% improvement over the naive predictor.

However, I think the cross val score (across the entire dataset) better represents the model's performance. Here, the score is a more modest 0.449, which represents a 13.5% improvement over the naive predictor.

Sensitivity Analysis

I performed a sensitivity analysis by randomly selecting a set number of features, and replacing the values with random numbers chosen from a normal distribution. This was the first machine learning project in which I've done this, and it was an interesting learning experience.

The results are essentially what I expected; the model's performance declined steadily as I randomized more data. What I *didn't* expect was for my model to be so robust overall. It appears that the

SelectPercentile step of my pipeline is doing a good job of ignoring useless data.

Here are the results of my sensitivity analysis:

Condition	F-beta score
Final model	0.449
10 features randomized	0.455*
20 features randomized	0.449*
50 features randomized	0.438
100 features randomized	0.433
200 features randomized	0.418
400 features randomized	0.386
Naive model	0.396

* I'm not surprised that there's some noise and inconsistency when I randomize such a small subset of the data. I think these results mean that scrambling a tiny subset of the data has *no* effect on performance, not that it actually improves it!

Justification

The mean f-beta score when cross validating over my entire data set is 0.449, with a standard deviation of 0.028. This is with 4 stratified folds.

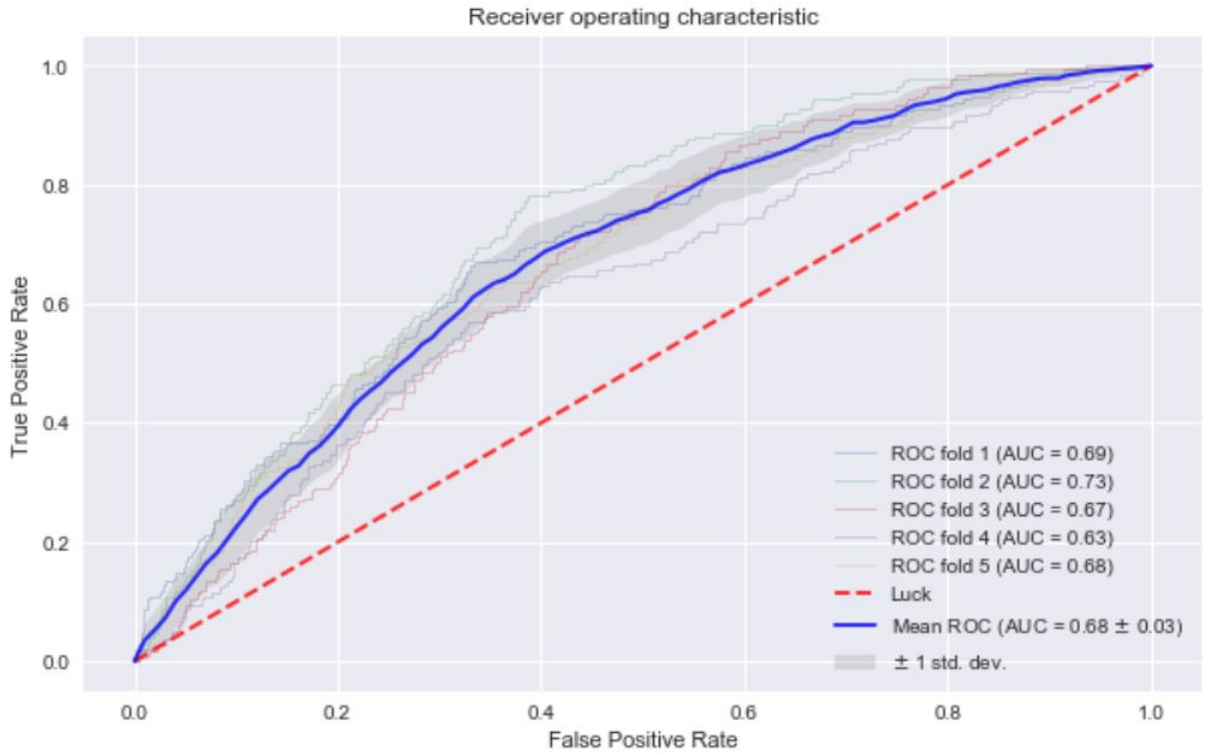
When we compare this to my naive predictor, we see a 13.5% improvement (0.449 versus 0.396).

Considering that we're predicting romantic attraction (a black-box process if there ever were one!), I'd consider this to be pretty good. However, I think it's a bit premature to say we've "solved" the problem.

V. Conclusion

Free-Form Visualization

Andrew Ng strongly advises ML engineers to pick a single metric to evaluate and stick with that throughout the entire process. I've used f-beta as that metric, but I still think it will be useful to visualize the model's performance using ROC-AUC. This allows us to see its performance in 2 dimensions instead of 1. So let's look at a comparison of the false positive vs. true positive rate:



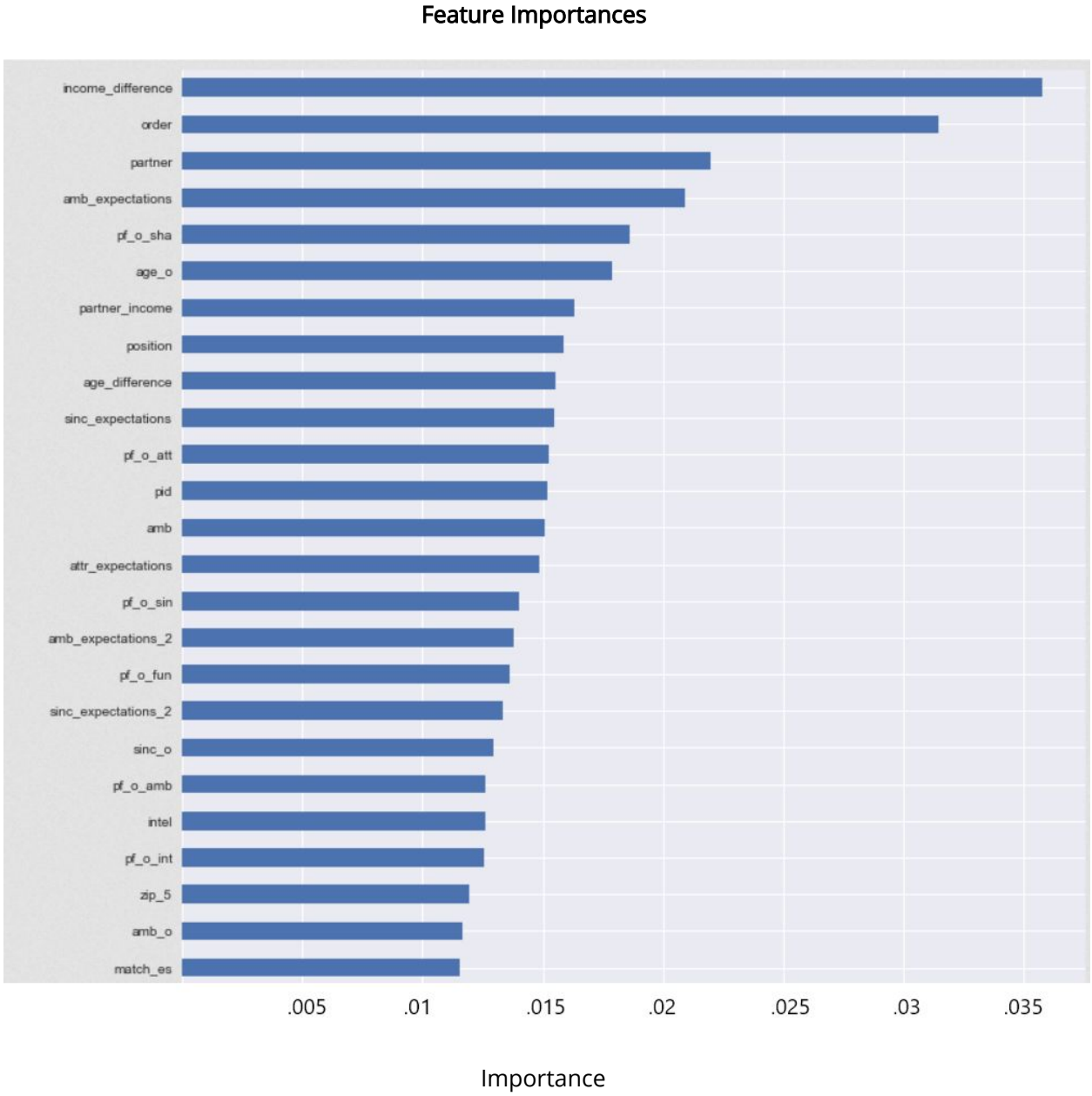
The dotted red line represents the romantic connections we would find due to chance, or just selecting pairs randomly. The blue line represents our model's performance, and we see that it performs quite well.

One of my data science mentors suggests looking at the true positive rate when the false positive rate equals 50%.

- A true positive rate above 70% can be considered "good,"
- A true positive rate above 80% is "really good," and...
- A true positive rate above 90% is "too good" -- your data might be leaking information from the future.

Mine scores around 75%, so it's actually a pretty good model by ROC standards!

Next, let’s examine the feature importances:



This is a cropped version of the graph, and the full version appears in the notebook. We've discussed the results a bit already; my engineered feature of log income difference was the most important, while "order" and "partner" were surprisingly important as well. We also see that several of the "expectations" features I created proved useful as well.

Reflection

I entered Udacity's machine learning program with about a year's worth of experience. I've been working on dozens of machine learning projects over the past year while continuing to teach psychology.

With that said, this is probably the hardest I've ever worked on a single project (I do have one other that's comparable)!

The data was really difficult to wrap my mind around, and I still need to constantly check the key to see what the feature names refer to. This was a great dataset to choose for the project; it was not only a topic I was interested in, but it was also complicated enough that I really had to apply myself to create a working machine learning model.

The process of feature engineering was surprisingly straightforward. Some of the coding aspects were a little challenging, but for the most part, I had a clear idea of how I could augment my data and Python was fairly cooperative.

I used just about every machine learning strategy I knew of in this project, even if some of them didn't make it into my final draft. I engineered features, transformed the data, tried PCA, tried creating synthetic data, used feature selection and grid search, and then even created my own custom classifier (something I had no idea how to do before this).

The custom classifier was probably the most valuable lesson I learned; it taught me that "turning the knobs" may only result in a modest performance increase; if you really want a great model, you have to fundamentally reconsider how you handle your data. This includes how you collect it, as well as how you make predictions and measure performance.

Here's a good analogy. Last year I was looking for a new roommate, and created an application to screen people for conscientiousness (how clean, organized, and responsible they were). But as soon as I imported the personality test, which contains questions like "How often do you ignore your responsibilities?" I realized I can cut to the chase and just ask questions like "How often do you leave dishes in the sink?"

This is the importance of gathering the right data and asking the right questions.

In terms of my grid search, this was definitely the hardest I've ever worked to optimize a model. I ran several searches to get a feel for what was working and what wasn't, but it culminated with an exhaustive search that ran continuously for about 2 days. I really did eke out as much performance as I could!

But, for the reasons I've discussed earlier, I wouldn't yet use this model in any sort of "production" environment -- whether it were my own speed dating service or a dating app. I think it's off to a really good start -- and the project certainly shows off my machine learning chops -- but the features probably aren't the best ones for predicting romantic connections.

Improvement

I think I could improve the performance significantly if I had a say in the data gathered. A lot of the data is redundant, too unstructured, or doesn't truly touch on the factors that contribute to romantic compatibility.

My psychological expertise is in personality theory, so I can provide a brief overview of the direction I'd take this data. In psychology, we've found that there are 5 key dimensions of personality -- think of them as elements on the periodic table. We refer to them as the Big Five, and they can be summarized as:

- **Extraversion**. High scorers are cheerful and enthusiastic; low scorers are aloof and quiet.
- **Agreeableness**. High scorers are kind, generous, and cooperative; low scorers are hostile, judgmental, and combative.
- **Conscientiousness**. High scorers are organized and responsible; low scorers are impulsive and irresponsible.
- **Neuroticism**. High scorers are moody and anxious; low scorers are calm and emotionally stable.
- **Openness to experience**. High scorers are curious, creative, and intellectual; low scorers are old-fashioned and conservative.

I believe that properly measuring these five traits would massively improve the quality of the model. We tend to associate with people that are similar to ourselves. I'd be surprised to see a conscientious honor roll student dating a convicted felon, or an eccentric artist dating a construction worker. The Big Five don't determine our destiny, but this data would filter out a lot of obviously bad matches.

Furthermore, I'd want to gather additional data on one's lifestyle and values:

- Specific religion, and how important it is to them
- How often someone smokes and drinks (although this is strongly predicted by conscientiousness!)
- Political opinions (which is moderately predicted by openness)

I would be very confident in my model if I had this sort of data available to me.

There are few more ideas -- more from the machine learning side -- that could have possibly improved performance.

I didn't want to include features that were too similar to the match variable -- for instance, "How much do you like this person?" But I think I

could have repurposed this data in a way that's more similar to current dating apps.

Here's what I could have done. I could have calculated each person's "attractiveness score" by using `pd.DataFrame.groupby()`, and then compared these scores between participants and their partners. A big disparity would almost certainly mean there wasn't a match. Services like OkCupid already do this by counting the number of "likes" and messages each profile gets. They then allow premium members to sort by attractiveness, as rated by fellow users. This would have been a really interesting strategy to try!

Lastly, the one ML technique that I know I could have used, but didn't, is stacking classifiers. There are libraries like *mlxtend* that allow you to fit several models to your data, and then use a "meta-model" to aggregate predictions and correct potential errors between them. I've been experimenting with this technique lately, but the grid search took so long as it was that I simply didn't have the time or patience to stack multiple classifiers on top of each other. I expect this would have only improved performance by 2% at best, based on my previous experience with them.

I've really enjoyed working on this project, and I hope it will sufficiently demonstrate that I'm skilled at using data science to predict human behavior!