Speed Dating Analysis and Prediction

AUTOMATIC LEARNING 1

DATA SCIENCE AND ENGINGEERING
POLYTECHNIC UNIVERSITY OF CATALONIA

Alina Castell Blasco - 49188689S
Silvia Vallet Marginet - 26891402D

June 2023

Contents

1 Introduction

Over the years, humans have continued to evolve being driven by one basic instinct: reproduction. That is, in fact, the basis of the survival of humankind and the reason why, at some point in our lives, we might find ourselves looking for a partner.

However, this couple-seeking process has developed just as much as society has: from courtship to dating apps, partner choosing as we know it is nothing like it used to be. But behind every date, or every *Tinder match*, could there still be a pattern? Could our partner choices be in fact much more predictable than we believe them to be?

Let us use Machine Learning to find out.

Our purpose is to see if we can predict whether two separate individuals will fancy each other or not, based on our prior knowledge about them (including, but not being limited to, traits such as gender, age, race, studies...). In short, we will answer the question: Can we predict attraction through data?

To do so, we will be working on the dataset *Speeddating*, which was released in 2004 by Ray Fisman and Sheena Iyengar from the Columbia Business School, gathering the data from a series of speed dating events from 2002-2004. The dataset file was obtained from the OpenML repository, which can be checked in the Reference section.

In these 4 minutes dates, participants were asked to rank their dates based on several attributes and to conclude whether they would consider them to be a match or not.

More specifically, the focus of this paper is to determine whether attributes such as gender or race are influential when it comes to choosing a partner and, most importantly, what aspects are most valued in a partner.

2 Previous work

2.1 From *.arff* to *.csv* format

After downloading the *Speeddating* dataset from *OpenML*, we notice it is in *.arff* format right away. Therefore, as we will be working with *Pandas*, a *Python* data analysis library, we shall read it as an *.arff* and then convert it into a *Pandas* dataframe.

```python
# reading the dataset
raw_data = loadarff('speeddating.arff')
# naming our dataset 'SD' as for 'Speed Dating'
SD = pd.DataFrame(raw_data[0])
```

Listing 1: Loading the dataset

We are now able to peak into the data using *Pandas* commands `SD.head()` or by simply printing it.

However, one issue that slightly bothered us was the way categorical features were in *textiowrapper* format, rather than as a regular *string*. To prevent this from causing future computational issues, we fixed it (Listing 3).

```python
# take categorical features
to_change = SD.columns[SD.dtypes=='O']
# remove b'' characters
for col in to_change:
  for i in range(len(SD)):
    SD[col][i] = str(SD[col][i])[2:-1]
```

Listing 2: Cleaning the dataset

We are now able to save this new data frame in *.csv* format to avoid repeating this process in future executions with the command `SD.to_csv('speeddating.csv',index=False)`, that finishes clearing the dataset.

3 Data exploration

Taking a first glance at our data, we identify our features and observations. We soon recognize *match*, which will be our target variable. Our entire analysis will revolve around its value as we want to predict if a couple will end up being a match or not.

The complete list of features and their meanings can be found in Listing 8 (Appendix).

Some quick remarks:

- Features containing suffix *'_o'* assess, not self traits, but partner traits (e.g: *age* : age of self, *age_o* : age of partner). Not to be confused with traits containing suffix *'_o'* referring to partner's rating on self (e.g: *'attractive'* : rating of self's attractiveness, *'attractive_o'*: partner's rating on self's attractiveness).
- Features containing either prefix *'importance_'* or suffix *'_important'* assess self preferences, importance given by self to certain qualities in a partner (e.g: *'attractive_important'* : self's rating on importance of attractiveness in a partner).
- Features containing prefix *'pref_o_'* assess, not self's, but partner's personal preferences (e.g: *'pref_o_attractive'* : partner's rating on importance of attractiveness).
- Features containing suffix *'_partner'* assess self's rating on partner traits (e.g: *'attractive'* : rating of self's attractiveness, *'attractive_partner'* : self's rating on partner's attractiveness).

In brief, as the list of features seemed quite long, we assessed the size of our data frame to see what we were really up against (command `SD.shape`). We then discovered our dataset to be incredibly large, with a total of 8378 rows (observations) and 123 columns (features).

As a matter of fact, we had to re-adjust *pandas* so that all our features would show during the execution process, and not just a couple of them (Listing 3).

```
pd.options.display.max_columns = 999
pd.options.display.max_rows = 999
```

Listing 3: Adjusting *pandas*

Thus, although we still had not dealt with missing values, and as substantial as working with all this data could have been, we were strongly advised by our lecturer to remove some of our features, based on certain criteria, so as to reduce our data frame's size. This is where the real challenge began.

3.1 Resampling

Prior to any data processing, we shall always split our dataset into a train set and a test set, so as to avoid any contact between the two sets as we apply transformations onto our data. Furthermore, a validation set will also be generated for the ensemble methods that will be explained later on.
The code for this dataset definition can be found in the Annex.

3.2 Pre-processing

3.2.1 Feature extraction

Our data set originally contained 123 features and over 8000 observations, which is indeed big. This left us with no choice but to discard some features:

- Firstly, we decided to remove *'has_null'* as it was superfluous for our analysis.
- Similarly, since our purpose is to predict whether a couple ends up being a match or not with our target feature being *'match'*, leaving columns *'decision'* and *'decision_o'* made no sense as, combined together equal *'match'*, as shown in Figure 1. In fact, we fitted several models with these two features alone and consistently got an accuracy of 100%, which means that including them is absurd.
- Lastly, we removed *'field'* has its encoded version had 56 categories.

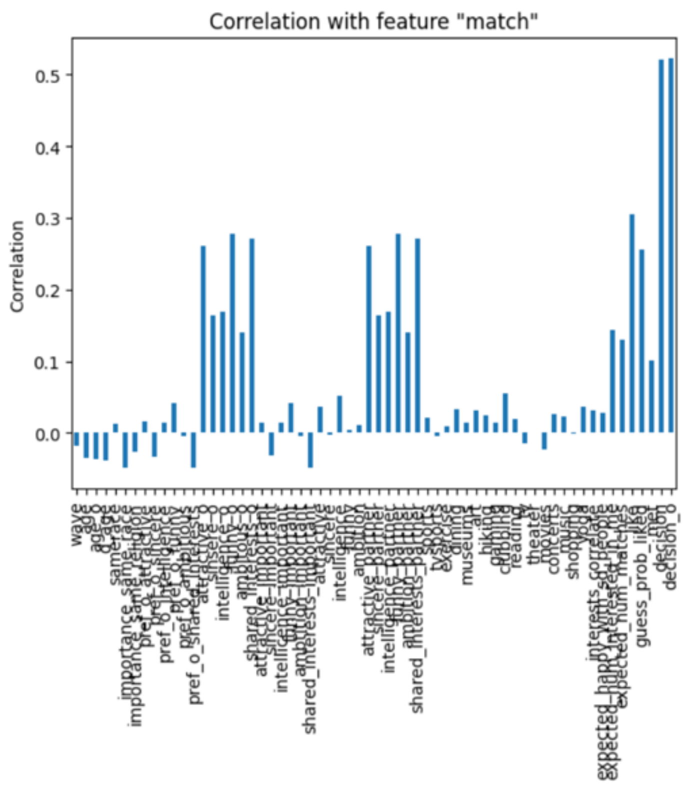


Figure 1: Correlation of our features with our target *'match'*

3.2.2 Treatment of mixed Data Types

We then assessed our features' type and established that our data set contained both numerical and categorical features by using the command `SD.types`.

On the one hand, we had up to 60 categorical variables (originally 64). Coincidentally, none of these features seemed to have missing values.

On the other hand, we had 59 numerical variables. Both shown in (Listing 4).

```
SD.dtypes[SD.dtypes == 'object'].count()
SD.dtypes[SD.dtypes == 'float64'].count()
```

Listing 4: Total of categorical features

However, to our surprise, some of these numerical variables explain ratings (e.g: 0-10 scores). Thus, these variables are in fact categorical and must be labeled as such.

So as to decide whether a variable is categorical or not, we may compute their histogram (Listing 5).

```
import seaborn as sns
sns.histplot(data=SD, x='attractive')
```

Listing 5: Distribution of a feature

For instance, Figure 2, shows a distribution typical of a categorical variable. However, the variable in question, *attractive*, was originally labeled as a numerical one, so we will change it.

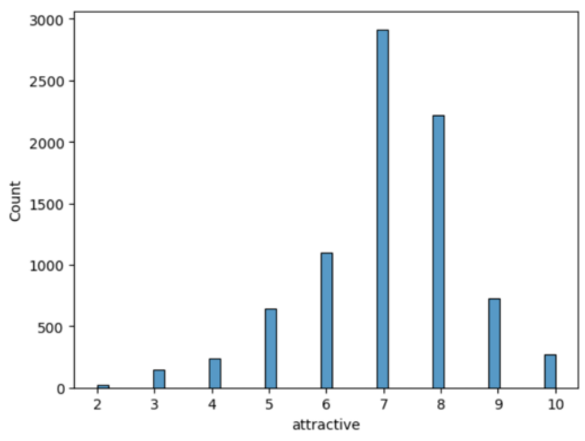


Figure 2: Histogram for feature *'attractive'*

On the contrary, Figure 3, depicts a distribution usually seen among continuous variables. Although the variable behind this distribution, *attractive_important*, is actually a rating, this time, making it categorical would not make any sense.

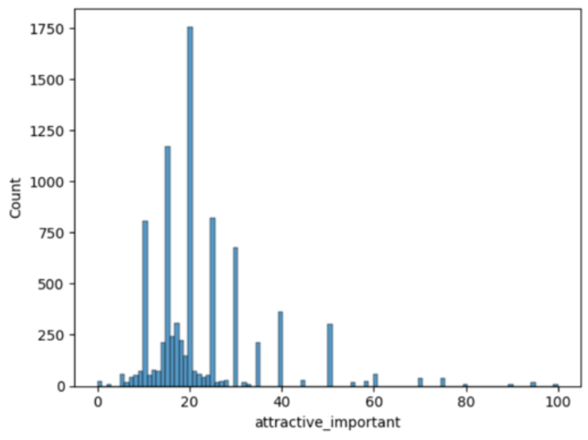


Figure 3: Histogram for feature *'attractive_important'*

In some cases this classification was somewhat confusing as there were distributions which were close to categorical but had values in between that did not allow us to label them as such (Notice the short bars in range [6-11] shown in Figure 4).

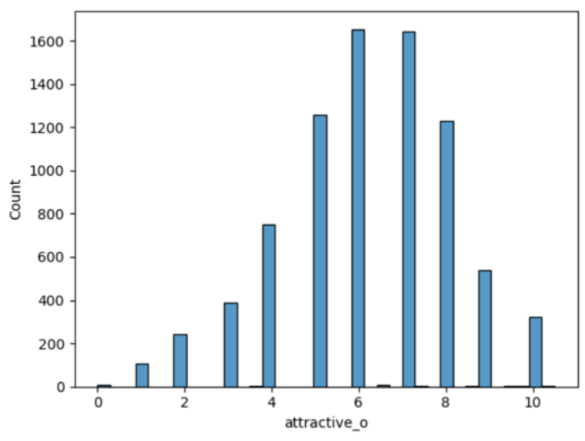


Figure 4: Histogram for feature *'attractive_o'*

Finally, proceeding according to this reasoning, the list of falsely numerical variables that are in fact categorical includes: 'expected_happy_with_sd_people', 'importance_same_race', 'importance_same_religion', 'd_importance_same_race', 'd_importance_same_religion', 'attractive', 'sincere', 'intelligence', 'funny', 'ambition', 'sports', 'tvsports', 'exercise', 'dining', 'museums','art', 'hiking', 'gaming', 'clubbing', 'reading', 'tv', 'theater', 'movies', 'concerts', 'music', 'shopping', 'yoga', 'met'.
We now change their *dtype* from numerical (*'float64'*) to categorical (*'object'*) manually (Listing 6).

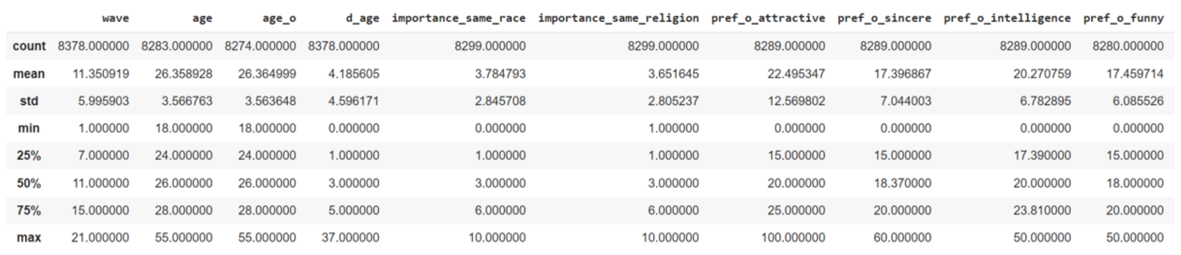
```
false_numericals = ["expected_happy_with_sd_people",
    "importance_same_race","importance_same_religion","d_importance_same_race",
    "d_importance_same_religion", "attractive", "sincere", "intelligence",
    "funny", "ambition", "sports", "tvsports", "exercise", "dining",
    "museums", "art", "hiking", "gaming", "clubbing", "reading", "tv",
    "theater", "movies", "concerts", "music", "shopping", "yoga", "met"]
SD[false_numericals] = SD[false_numericals].astype(object)
```

Listing 6: Re-labeling false numerical variables

After re-labeling these variables, we are now down to 86 categorical features (originally 90) and 33 numerical features.

3.2.3 Dealing with Missing Values

By taking a first glance at our data frame's distribution (using the command `SD.describe()`), it can be easily seen that the mean, minimum and maximum for most variables is in range with what we would have expected. It also seems that some variables rating or punctuating partners, rate out of 10, while others out of 100. This will probably call for future normalization. We further noticed that most features had missing values among their many observations, as the total 'count' was under 8378, as depicted in Figure 5.



	wave	age	age_o	d_age	importance_same_race	importance_same_religion	pref_o_attractive	pref_o_sincere	pref_o_intelligence	pref_o_funny
count	8378.000000	8283.000000	8274.000000	8378.000000	8299.000000	8299.000000	8289.000000	8289.000000	8289.000000	8280.000000
mean	11.350919	26.358928	26.364999	4.185605	3.784793	3.651645	22.495347	17.396867	20.270759	17.459714
std	5.995903	3.566763	3.563648	4.596171	2.845708	2.805237	12.569802	7.044003	6.782895	6.085526
min	1.000000	18.000000	18.000000	0.000000	0.000000	1.000000	0.000000	0.000000	0.000000	0.000000
25%	7.000000	24.000000	24.000000	1.000000	1.000000	1.000000	15.000000	15.000000	17.390000	15.000000
50%	11.000000	26.000000	26.000000	3.000000	3.000000	3.000000	20.000000	18.370000	20.000000	18.000000
75%	15.000000	28.000000	28.000000	5.000000	6.000000	6.000000	25.000000	20.000000	23.810000	20.000000
max	21.000000	55.000000	55.000000	37.000000	10.000000	10.000000	100.000000	60.000000	50.000000	50.000000

Figure 5: Partial results for *SD.describe()*

So as to identify the features with the most missing values, we looked for those features containing over 1000 missing values. We then found that:

- Feature *'expected_num_interested_in_me'*, containing 6578 missing values.
- Feature *'expected_num_matches'*, containing 1173 missing values.
- Feature *'shared_interests_o'*, containing 1076 missing values.
- Feature *'shared_interests_partner'*, containing 1067 missing values.

That said, although *shared_interests_important* and *pref_o_shared_interests* had ten times less missing values, leaving them was quite questionable as they referenced some of the features we just deleted.

For the rest of features, we used the *.isna* command along with feature *'has_null'* (indicator variable for missing values in observations, eventually removed as it is of no use for the analysis) to identify the remaining missing values.

Our first idea was imputing these values through *K-Nearest-Neighbour* (KNN). This method looks at the neighbors of the missing values and imputes their values. The only issue was the fact that KNN relies completely on continuous features. Thus, as we had only 2 remaining numerical variables free of missing values, it seemed unreasonable to impute the rest of missing values based on just 2 features.

Thus, we will did not perform KNN to impute our missing data.

Nonetheless, as features with an extraordinary amount of missing values had already been removed, it was time to look for missing values not in features but in observations. More specifically, all those observations (rows) containing missing values were dropped (`SD = SD.dropna()`). This reduced our total of observations from 8378 to 6503. Which, although smaller, could still be considered a large amount of observations.

It must be kept in mind that applying the *.isna()* command directly, without removing features with most missing values first, would have reduced our data set observation's to $\frac{1}{8}$ its size, which was not reasonable. This is mainly due to the fact that feature *'expected_num_interested_in_me'* contained 6578 missing values and, unless the variable is removed first, *.isna()* disregards 6578 observations.

3.2.4 Fixing feature *'met'*

By first looking at our data we discovered an anomaly concerning feature *'met'* (*Have you met your partner before?*). This feature was supposed to be a *yes* or *no* categorical feature, specially since it mostly took 0 and 1 as values. However, when executing the command `SD.met.value_counts()` we discovered that, for unknown reasons, feature *'met'* took:

- Value 7.0 in up to 3 observations.
- Value 5.0 in up to 2 observations.
- Values 3.0, 6.0 and 8.0 in 1 observation (each).

In view of this, we pondered whether replacing these values with '1' or with 'NAN' (Listing 7) would make more sense.

```
# Replacing values greater than 1 with 1
SD.loc[SD['met'] > 1, 'met'] = 1
# Replacing values greater than 1 with NANs
SD['met'] = SD['met'].apply(lambda x: x if x <= 1 else None)
SD.dropna(subset=['met'], inplace=True)
```

Listing 7: Replacing values

Assuming that perhaps the variable could be referring to the number of times they had met before, we chose the first option. Either way, the difference between these two approaches is superfluous as we are talking about 8 observations when our total of observations is over 6000.

3.2.5 One-hot-encoding

In order to later perform several procedures such as *Lasso Regression* or *LDA*, our categorical features need to be encoded into numerical ones. This is done through *One-Hot-Encoding* code that can be checked in the Listing 10.

After performing it, we were down to 435 categorical columns (458 in total).

The output obtained is a new data set with six times more variables that represent each category of each original variable. The next step is to group these new variables as a category of their original one, the names assigned to the new ones are 'race_0', 'race_1', 'race_2' or 'gender_0' and 'gender_1' for example for the variables 'race' and 'gender', respectively. To understand and interpret this new data frame, its header must be recomputed with the code in Listing 11.

This way, by looking at our new distribution of categorical variables, we found that there are:

- 3 binary variables $[0-1]$: *'gender'*, *'samerace'* and *'met'*.
- Variables using 3 categories: variables starting with either prefix *'d_'* or *'d_pref_o_'* (which explain intervals).
- 1 feature using 4 categories: 4 *'d_d_age'* (there are 4 groups of age).
- 1 feature using 5 categories: *'race'* (there are 5 races).
- Ratings [0-10]: *'importance_same_religion'*, *'importance_same_race'*, *'expected_happy_with_sd_people'*, all hobbies (*'sports'*, *'tvsports'*, *'exercise'*, *'dining'*, *'museums'*, *'art'*, *'hiking'*, *'gaming'*, *'clubbing'*, *'reading'*, *'tv'*, *'theater'*, *'movies'*, *'concerts'*, *'music'*, *'shopping'*, *'yoga'*) and all personal traits (*'attractive'*, *'sincere'*, *'intelligence'*, *'ambition'* and *'funny'*).

3.2.6 Normalization

One must keep in mind that our numerical features had different ranges and required standardization, as seen in Figure 6.

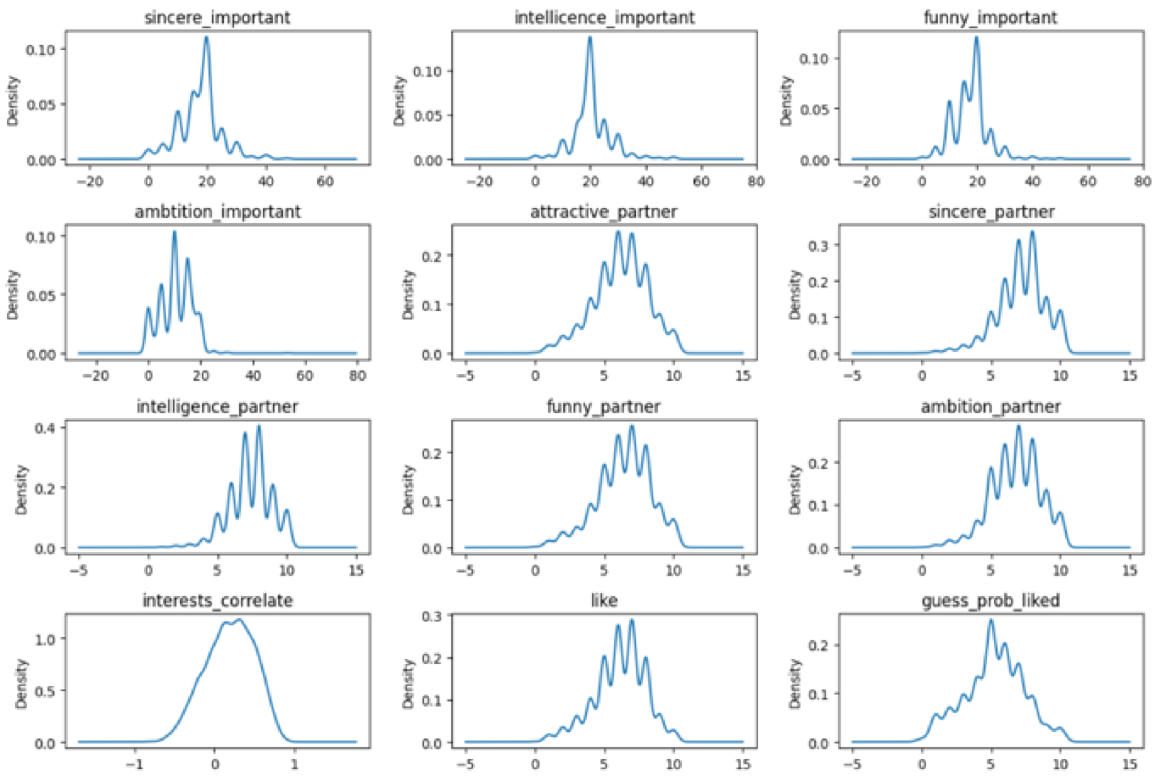


Figure 6: Distribution of some numerical features

Thus, we will be using *Min-Max scaling*: $\frac{X-X_{min}}{X_{max}-X_{min}}$, which will send our continuous data to the range [0-1].

However, the scaling process needs to be different for the training and test set so as to make things compatible with the fact that we cannot use information from the test set to do the scaling (Listing 12).

Figure 7 depicts how ranges have been normalized.

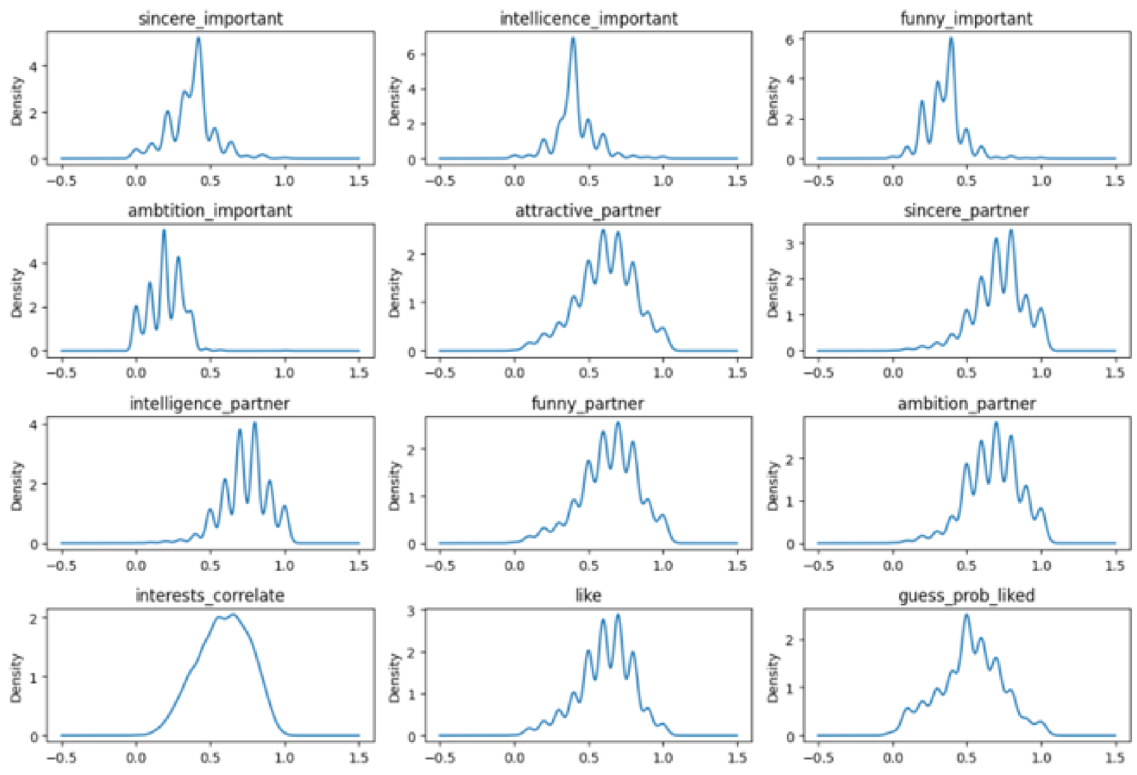


Figure 7: Distribution of some numerical features after standarization

4 Linear Classification

In this section we will first perform linear classification on our clean dataset. Next, we will conduct a feature selection process to compare our original linear classification results with the ones obtained working with a smaller subset of data (most important features).

More specifically, the linear classification methods we will apply will be Linear Discriminant Analysis (LDA), Quadratic Discriminant Analysis (QDA), K-Nearest Neighbour (KNN), Gaussian Naive Bayes (GNB) and Logistic Regression. Additionally to the methods seen in our course, we will further use Support Vector Classifier (SVC), Perceptron and Gradient Boost (GB).

4.1 Linear Classification

First, we will apply linear classification to the preprocessed dataset and we will perform a study of all possible classifiers and compare their accuracy values.

4.1.1 Linear Discriminant Analysis (LDA)

The Lineal Discriminant Analysis method is a generalisation of the Fisher's lineal discriminant. More specifically, LDA tries to model the probability $p(y = C_k|X = x)$ by assuming:

- $p(x|C_k)$ is Gaussian (which means that can be described by μ_k and Σ_k)
- All covariance matrix are the same ($\Sigma_k = \Sigma$)

By using the Bayes formula ($p(A|B) = \frac{P(B|A)P(A)}{P(B)}$) and all these assumptions, we obtain the next discriminant function: $a_k(x) = x^T\Sigma^{-1}\mu_k - \frac{1}{2}\mu_k^T\Sigma^{-1}\mu_k + log(\pi_k)$, where π_k are the prior probabilities.

With the commands in Listing 13 we fit a model through LDA and compute cross-validation metrics for this method. The output obtained are the global priors and means for each variable, as well as the method's metrics:

```
    Accuracy F1  Precision  Recall
LDA 0.83989 0.678842 0.716138 0.658797
```

Based on our observations, we have determined that the method exhibits a notable level of performance due to its high accuracy value. However, it falls short of its full potential as indicated by the relatively low F1 score.

4.1.2 Quadratic Discriminant Analysis (QDA)

Quadratic Discriminant Analysis is very similar to LDA. The main difference is that, in this model we do not assume that all the classes have the same covariance matrices. This leads to obtaining a quadratic decision surface. This model can also be regularized with its regularization parameter.

Since the code for every method has few variation, we will not repeating them completely, only the variations themselves. In this case is the command `qda_model = QuadraticDiscriminantAnalysis (reg_param=0.1).fit(X_train, y_train)`.

This table is the output of the accuracy for this method.

```
    Accuracy F1  Precision  Recall
QDA 0.70357 0.612559 0.611746 0.675554
```

We observe that the accuracy worsens with QDA.

4.1.3 K-Nearest Neighbors (KNN)

The K-Nearest Neighbors (k-NN) algorithm utilizes the training set samples as reference points to classify new samples. When a new sample is encountered, k-NN compares its attributes with those of the training set and assigns it to the class that has the highest similarity based on a specific distance metric. The k value determines the number of training set samples considered for comparison. A larger k value means more samples will be used in the decision process. To determine the optimal k value and distance metric for our data, we will employ grid search cross-validation.

The code for this method can be found as well in the Annex section as the Listing 14.

We will take k=1 and distance = 'euclidean' and add k-nn to our results table.

```
    Accuracy F1  Precision  Recall
KNN 0.832799 0.514668 0.765001 0.530212
```

4.1.4 Gaussian Naive Bayes (GNB)

Naive Bayes makes the assumption that the attributes follow a certain distribution and are conditionally independent given the class. Specifically, for Gaussian Naive Bayes, when working with numerical variables, it assumes that the features are independent of each other and follow a Gaussian (normal) distribution. This means that the likelihood of each feature value given the class is modeled using a Gaussian distribution.

This is the output obtained when applying this method:

```
    Accuracy F1  Precision  Recall
Gaussian Naive Bayes 0.332574 0.33236 0.573983 0.575512
```

This method is by far the worse one.

4.1.5 Logistic Regression

The logistic regression algorithm applies a logistic function, which allows us to analyze the relationship between variables and make predictions based on probabilistic outcomes. To fit this logistic regression model, the algorithm estimates the coefficients or weights of the independent variables through the maximum likelihood estimation. It aims to find the optimal values of these coefficients that maximize the likelihood of the observed data. The code used for this method can be found in Listing 15.

The result table is:

```
    Accuracy F1  Precision  Recall
Logistic Regression 0.843928 0.666933 0.728928 0.642242
```

The result is considerably good overall.

In addition to these previous linear classifiers that have been studied in the course, we have decided to apply the following three linear classifiers in order to obtain a wider view of the classification. This way we will be able to find out if there are better classifers.

4.1.6 Support Vector Classifier (SVC)

This algorithm separates data points into different classes, suring the training process, the SVC determines the optimal hyperplane that best separates the data points based on their features. This hyperplane is chosen to maximize the margin and minimize the classification error.

```
    Accuracy F1  Precision  Recall
SVC 0.844921 0.599468 0.778583 0.581921
```

4.1.7 Perceptron

This algorithm learns a linear decision boundary based on the input features. During the training process, the Perceptron adjusts its internal weights based on the input features and their corresponding target labels. It iteratively updates the weights to minimize the misclassification error until convergence or a maximum number of iterations. Note that the perceptron algorithm is only suitable for linearly separable data.

```
    Accuracy F1  Precision  Recall
Perceptron 0.783148 0.648093 0.67813 0.678911
```

4.1.8 Gradient Boost

This algorithm is a type of ensemble learning method that sequentially builds new models to correct the mistakes made by previous models. Gradient Boosting is known for its ability to handle complex relationships in the data and achieve high predictive accuracy. During the training process, the algorithm fits a series of weak prediction models to the data. Each new model is trained to minimize the loss function by correcting the mistakes of the previous models.

```
    Accuracy F1  Precision  Recall
Gradient Boost 0.857277 0.697402 0.767372 0.667754
```

4.1.9 Generalisation Performance

Overall, this is the final table with comparisons of all the linear classification methods studied:

	Accuracy	F1 Macro	Precision Macro	Recall Macro
Gradient Boost	0.857277	0.697402	0.767372	0.667754
Logistic Regression	0.849266	0.640662	0.765409	0.613789
SVC	0.844921	0.599468	0.778583	0.581921
LDA	0.83989	0.678842	0.716138	0.658797
KNN	0.785911	0.624472	0.625317	0.625072
Perceptron	0.783148	0.648093	0.67813	0.678911
QDA	0.70357	0.612559	0.611746	0.675554
Gaussian Naive Bayes	0.332574	0.33236	0.573983	0.575512

Table 1: Comparison of linear classification methods ordered by accuracy.

In a nutshell, Gradient Boost and Logistic Regression seem to be the best possible models overall, with a respective accuracy of 85.7% and 84.9%, closely followed by SVC, whereas Gaussian Naive Bayes is clearly the worst model. Since Gaussian Naive Bayes is that much worse than the rest, this could be indicating that there is over fitting. We will discuss this later.

With this being said, we shall choose Logistic Regression, as best model out of those seen during our course, along with Gradient Boost, as best additional model discovered through our own research, to analyse generalization performance. We shall fit this model to analyse generalization performance. We have computed the prediction for the gradient boost as well; nevertheless, since it is not of interest we do not show the output here, it can be found in the attached notebook. We obtain the following table:

```
predicted 0.0 1.0
target
0.0 1631 144
1.0 215 141       precision    recall  f1-score   support

        0.0       0.88      0.92      0.90      1775
        1.0       0.49      0.40      0.44       356

   accuracy                           0.83      2131
  macro avg       0.69      0.66      0.67      2131
weighted avg      0.82      0.83      0.82      2131
```

This table is obtained by running the command found in Listing 16 which generates a confusion matrix of the correctly predicted instances (True positive and negative) and the wrongly predicted ones (False positive and negative). In addition, it generates a classification report showing the precision, recall, F1 score and support of the instances, as well as their averages and accuracy. We will analyse the F1-score because it is the harmonic mean of precision and recall and it provides a single metric that balances both of them; as well as the accuracy and the confusion matrix. This type of table will be shown further in the study so same analysis will be applied.

The precision shows that out of all instances predicted as matches only 49% of them were actually match. The recall shows that the model is able to capture only a 40% of the actual matches present in the dataset. The F1-score for the positive class ('yes' to match) achieves a low balance between precision and recall of 44%. The accuracy value is considerably good since it is an 83%, meaning that it correctly predicts the outcome ('match' or not) for approximately 83% of the instances. As for the confusion matrix there are much more false positives, instances misidentified as 'match', than true positives, instances truly identified as 'match'. For the instances identified as not match the proportion is way better.

4.2 Feature selection/extraction

Although we had obtained quite good results working with some models such as *Logistic Regression*, the fact that we were getting such poor performance metrics from *Gaussian Naive Bayes* and that the difference, in terms of performance, between these two models mentioned was so big, led us to believe that there could perhaps be some over fitting, meaning that, additionally to the ones we had removed during our data's preprocessing, more features should be removed.

We then computed our feature's autocorrelation matrix to see if we had correlated features, but saw that they actually were not that correlated, as seen in Figure 8.

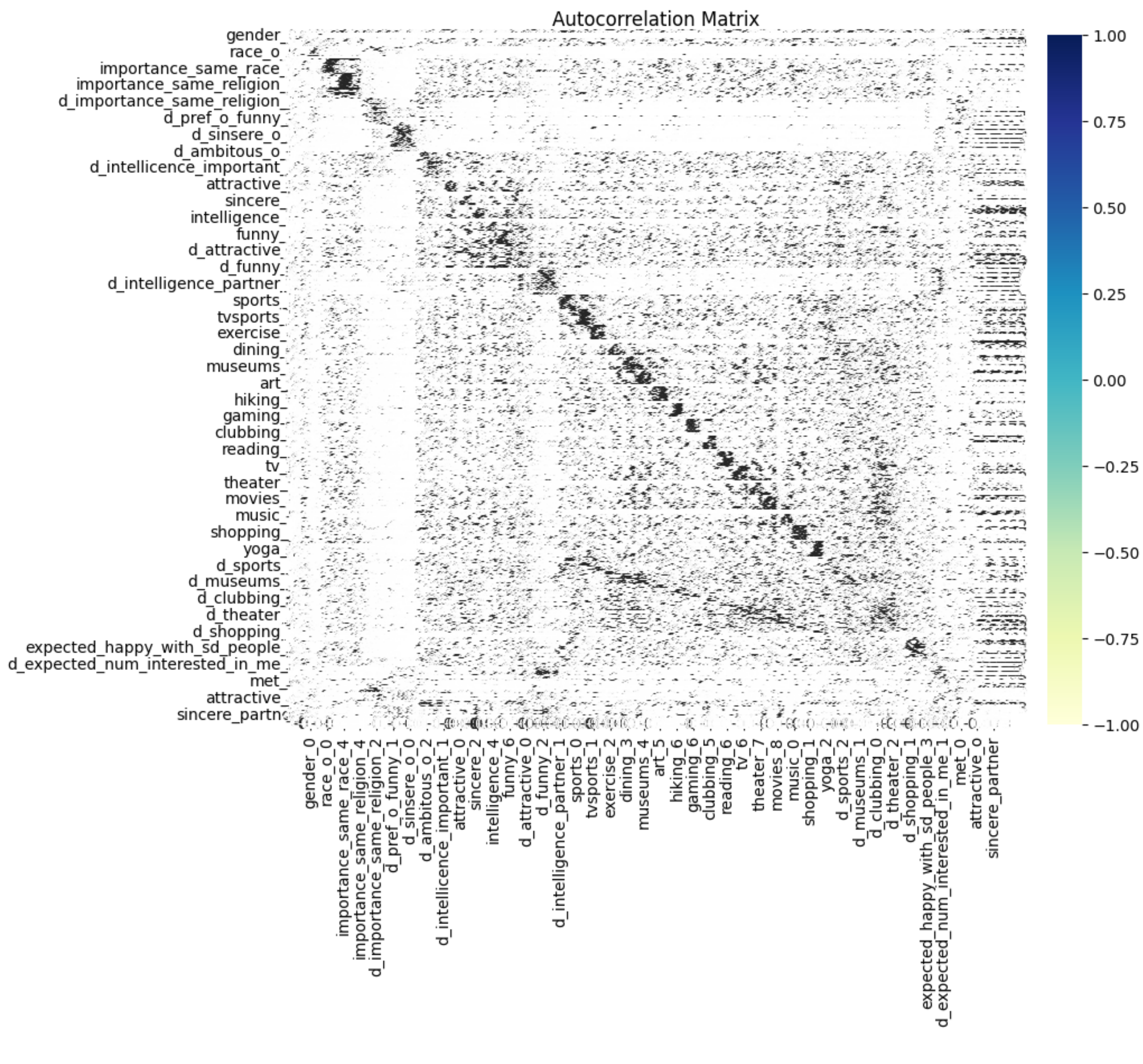


Figure 8: Feature's autocorrelation matrix

Therefore, as we cannot just choose which variables to remove at random, we want to disregard those that are shown to be truly insignificant. Therefore, we decided to select the most relevant features that will provide the best prediction based on the *Lasso Regression* method.

4.2.1 Lasso Regression

To do so, we will fit a model that will actually allow us to do feature selection (including categorical variables after *one-hot-encoding*) by performing *Lasso Regression* with *cross-validation.* The code can be checked at Listing 17

The 93 important columns identified by this method are:

’gender_0’, ’gender_1’, ’d_d_age_1’, ’d_d_age_3’, ’race_0’, ’race_2’, ’race_o_1’, ’race_o_2’,
’importance_same_race_1’, ’importance_same_religion_0’, ’importance_same_religion_1’,
’importance_same_religion_7’, ’d_importance_same_race_2’, ’d_importance_same_religion_0’,
’d_pref_o_attractive_1’, ’d_pref_o_attractive_2’, ’d_pref_o_sincere_2’, ’d_pref_o_intelligence_0’,
’d_pref_o_intelligence_2’, ’d_pref_o_funny_0’, ’d_pref_o_shared_interests_0’, ’d_attractive_o_0’,
’d_attractive_o_2’, ’d_funny_o_0’, ’d_funny_o_2’, ’d_ambitous_o_2’, ’d_shared_interests_o_0’,
’d_shared_interests_o_2’, ’d_attractive_important_2’, ’d_intellicence_important_0’,
’d_intellicence_important_2’, ’d_funny_important_0’, ’attractive_4’, ’attractive_5’, ’attractive_6’,
’sincere_4’, ’sincere_7’, ’sincere_8’, ’intelligence_4’, ’intelligence_5’, ’intelligence_6’, ’funny_5’, ’funny_6’,
’ambition_5’, ’d_attractive_1’, ’d_attractive_2’, ’d_intelligence_1’, ’d_intelligence_2’, ’d_funny_1’,
’d_ambition_1’, ’d_attractive_partner_0’, ’d_attractive_partner_2’, ’d_sincere_partner_0’, ’d_sincere_partner_2’,
’d_intelligence_partner_1’, ’d_funny_partner_2’, ’d_shared_interests_partner_0’, ’sports_0’, ’sports_2’, ’sports_3’,
’tvsports_1’, ’tvsports_2’, ’tvsports_3’, ’tvsports_4’, ’tvsports_7’, ’exercise_2’, ’exercise_6’, ’museums_4’,
’art_5’, ’hiking_1’, ’hiking_2’, ’hiking_6’, ’hiking_10’, ’gaming_2’, ’gaming_5’, ’gaming_6’, ’clubbing_0’,
’clubbing_5’, ’clubbing_7’, ’reading_5’, ’reading_7’, ’tv_4’, ’tv_6’, ’tv_7’, ’theater_7’, ’concerts_5’,
’music_6’, ’music_7’, ’shopping_1’, ’shopping_4’, ’shopping_5’, ’shopping_8’, ’yoga_1’, ’yoga_2’, ’yoga_4’,
’yoga_6’, ’d_sports_1’, ’d_exercise_2’, ’d_dining_1’, ’d_art_2’, ’d_movies_2’, ’d_concerts_0’, ’d_concerts_2’,
’d_shopping_0’, ’d_shopping_2’, ’d_yoga_0’, ’d_yoga_1’, ’d_interests_correlate_0’,
’expected_happy_with_sd_people_4’, ’expected_happy_with_sd_people_5’, ’expected_happy_with_sd_people_6’,
’expected_happy_with_sd_people_8’, ’d_expected_num_interested_in_me_0’, ’d_expected_num_interested_in_me_1’,
’d_expected_num_matches_0’, ’d_like_0’, ’d_guess_prob_liked_0’, ’d_guess_prob_liked_2’, ’met_0’, ’met_1’,
’attractive_o’, ’funny_o’, ’attractive_partner’, ’funny_partner’, ’like’, ’guess_prob_liked’.

So, now that the size of the data set used to actually predict the target variable has decreased, we will apply the classification methods once again so as to check whether this reduced data set is better in terms of prediction than the previous one or not.

4.3 Linear Classification with important features subset

We proceed to compare the different classification models again. This time, since the very same steps as before have been followed, we will not be discussing the procedure.

This is the final table with accuracy values for each method:

	Accuracy	F1 Macro	Precision Macro	Recall Macro
Gradient Boost	0.860021	0.700083	0.778402	0.668873
Logistic Regression	0.857962	0.716919	0.759236	0.692703
LDA	0.852701	0.69547	0.75015	0.669145
SVC	0.850867	0.634045	0.783324	0.607443
Perceptron	0.786154	0.680377	0.677017	0.718673
KNN	0.785911	0.624472	0.625317	0.625072
Gaussian Naive Bayes	0.772418	0.686878	0.670676	0.752233
QDA	0.767156	0.666635	0.652467	0.712437

Table 2: Comparison of linear classification methods ordered by accuracy.

Using this important feature subset set obtained through Lasso Regression, we have slightly improved our results in terms of model performance. Interestingly, the order in the ranking of models in terms of perfomance metrics remains similar. Although still far from the best, Gaussian Naive Bayes has perfomed way better this time, which supports our hypothesis that the previous dataset lead to over fitting. Again the Gradient Boost method is the best model overall but Logistic Regression has a very similar accuracy value.

4.3.1 Generalisation Performance with important features subset

Once again, we have chosen for prediction Logistic Regression as it is the best model studied in the course. We have computed the prediction for the gradient boost as well. Nevertheless, since it is not of interest we do not show the output here, it can be found in the attached notebook.

```
predicted 0.0 1.0
target
0.0 1652 123
1.0 219 137       precision    recall  f1-score   support

         0.0       0.88      0.93      0.91      1775
         1.0       0.53      0.38      0.44       356

    accuracy                           0.84      2131
   macro avg       0.70      0.66      0.68      2131
weighted avg       0.82      0.84      0.83      2131
```

This final result is similar as the one we had obtained prior to performing Lasso Regression, but slightly better. The precision shows that out of all instances predicted as matches 53% of them were actually match. The recall shows that the model is able to capture only a 38% of the actual matches present in the dataset. The F1-score for the positive class ('yes' to match) achieves a low balance between precision and recall of 44%. The accuracy value is considerably good since it is an 84%, meaning that it correctly predicts the outcome ('match' or not) for approximately 84% of the instances. As for the confusion matrix there are more false positives, instances misidentified as 'match', than true positives, instances truly identified as 'match'. For the instances identified as not match the proportion is way better.

Moreover, since the accuracy results are the somewhat similar for both data sets expect for Gaussian Naive Bayes, we will further apply the random forest method and check if its result is more conclusive than the previous ones in terms of which features are truly significant for our model.

5 Random Forest

One advantage that RF and DT have is that they perform feature selection on training.

For the train/validation datasets, we have used stratified partitions, the command used for the re-sampling is at the beginning of the study. This approach ensures that each partition maintains the same proportion of "yes" and "no" examples. It is particularly crucial when dealing with imbalanced datasets because a random partitioning could potentially worsen the imbalance if we are unlucky. As a result of the imbalance, the model may have a bias towards predicting the majority class (the one with more samples) more frequently, leading to poor performance in identifying the minority class (the one with fewer samples).

In this scenario, we will consider the misclassification errors equally important for both classes. To evaluate our model, we will use the following metrics: F1-score for class 1, F1-score for class 0, F1-score for average and accuracy.

5.1 Decision Tree

First, we will try to predict the target *'match'* with a decision tree which splits the data so that during the training it learns decision rules. The command used can be found at the Annex section in Listing 18 and its output is:

```
predicted 0.0 1.0
target
0.0 773 131
1.0 108 81         precision    recall  f1-score   support

          no       0.88      0.86      0.87       904
         yes       0.38      0.43      0.40       189

    accuracy                           0.78      1093
   macro avg       0.63      0.64      0.64      1093
weighted avg       0.79      0.78      0.79      1093
```

In this case, we can observe our data is highly imbalanced because for the validation sets there are 904 number of samples where [match = 'no'] and 189 number of samples where [match = 'yes']; we will take this into account in the following analysis. That is why the proportions of prediction are different for both outputs.

The precision shows that out of all instances predicted as matches only 38% of them were actually match. The recall shows that the model is able to capture only a 43% of the actual matches present in the dataset. The F1-score for the positive class ('yes' to match) achieves a low balance between precision and recall of 40%. The accuracy value is good since it is an 78%, meaning that it correctly predicts the outcome ('match' or not) for approximately 78% of the instances. As for the confusion matrix is quite proportional between the false positives, instances misidentified as 'match', and true positives, instances truly identified as 'match'. For the instances identified as not match the proportion is good as well.

Now we check if this model is overfitting by computing its train metrics (same command as before but with 'X_train' set) and checking its complexity (obtaining size of the tree). The output is:

```
predicted 0.0 1.0
target
0.0 2711 0
1.0 0 568
Tree depth: 16
Nodes: 761
```

It seems that the model is able to predict the training data perfectly, this means that we may be overfitting because our validation partition does not show the same results. In addition, we can say that it is a big tree since the number of nodes is considerably big and so is its depth. We will try to improve these results by optimizing its hyperparameters and recalculate the best decision tree with the optimum metrics.

The best set of hyperparameters and the accuracy values are:

```
{'criterion': 'entropy',
 'max_depth': 5,
 'max_features': None,
 'min_samples_leaf': 3,
 'min_samples_split': 2}
```

	Accuracy	F1-score (class 1)	F1-score (class 0)	F1-score (macro avg)
DT-best	0.84721	0.476489	0.910552	0.69352
DT-default	0.781336	0.40399	0.866106	0.635048

Table 3: Comparison of accuracy values in Decision Trees.

If we compare both trees, we can determine that the best decision tree has improven. However, we will apply the random forest method to find out if there is even a better solution.

5.2 Random Forest

The Random Forest algorithm is an ensemble method that combines multiple Decision Trees. Its purpose is to mitigate overfitting by averaging the predictions of individual trees, which are highly variable but uncorrelated.

In addition to this, Random Forest introduces the concept of Out-of-Bag (OOB) error. This error metric is be computed by evaluating the accuracy of the trees during the training process of the Random Forest model and serves a validation metric. The code for this method can be found in Listing 19.

As said, the OOB serves as a validation accuracy but we will also compute the real validation error, which code can be found in Listing 20.

With this we obtain:

```
predicted 0.0 1.0
target
0.0 2711 0
1.0 0 568        precision    recall  f1-score   support
```

```
          no       1.00      1.00      1.00      2711
         yes       1.00      1.00      1.00       568

    accuracy                           1.00      3279
   macro avg       1.00      1.00      1.00      3279
weighted avg       1.00      1.00      1.00      3279

OOB accuracy= 0.8539188777066179
Validation Accuracy:0.868252516010979
predicted 0.0 1.0
target
0.0 885 19
1.0 125 64
```

Both validation accuracy values are very similar and higher than the one obtained in the previous method. Nevertheless, opposite to the output of the decision tree we observe that the model is able to predict the training data perfectly, then we might be facing with overfitting. Firstly, we make sure that the problem is not related to the imbalance of our target; we recalculate metrics and apply RF again, but with no luck since this new accuracy is worse than the default one.

Secondly, we will tune our model hyperparameters in order to handle overfitting. The best parameters are:

```
{'class_weight': 'balanced_subsample',
 'max_depth': 100,
 'min_samples_leaf': 4,
 'min_samples_split': 6,
 'n_estimators': 200}
```

Once obtained the balanced RF and the best RF, we refit our model and obtain the following table comparing all methods.

	Accuracy	F1-score (class 1)	F1-score (class 0)	F1-score (macro avg)
RF-best	0.852699	0.570667	0.911099	0.740883
RF-default	0.868253	0.470588	0.924765	0.697677
DT-best	0.84721	0.476489	0.910552	0.69352
RF-balance	0.858188	0.387352	0.919814	0.653583
DT-default	0.781336	0.40399	0.866106	0.635048

Table 4: Comparison of Ensemble methods ordered by average accuracy.

Even though the accuracy value is maintained similar to the RF-default and RF-balanced, we can observe that the minority F1 score has improved greatly, up to 10 points more. Despite the high accuracy value of the Decision Tree approach, there is evidence that the Random Forest has the best accuracy and its minority F1 score has nearly 10 points more than the DT one. Also, the RF-bes is the only method that surpasses 50% bar on the F1 score which is a considerably good reason for choosing this method.

5.3 Feature importance

The feature importance of variables obtained from the best ensemble method is a valuable measure that helps identify the relevance of each variable in making predictions. However, the initial output without any filtering based on a specific threshold, such as the 'importance' value being greater than 0.005, might have been overwhelming due to the large number of variables. As a result, the first 41 most important features are now presented, ensuring that the focus is on the variables that have a notable impact on the predictions. The code used for generating this graphic can be found in Listing 21.

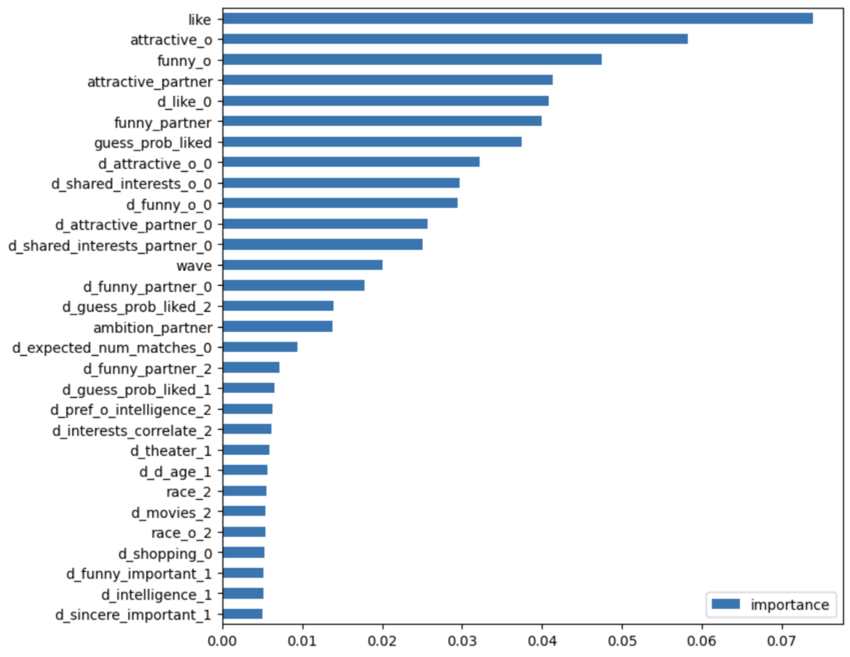


Figure 9: Sorted feature's importance.

5.4 Ensemble

We have a set of models, including some simple classifiers along with GaussianNB. To harness their collective power, we will combine them into a more powerful classifier using a voting approach. This voting classifier will utilize the predictions from each individual model to make its final prediction. By aggregating the outputs of multiple classifiers, we aim to leverage their diverse perspectives and improve overall performance.

We execute the same commands as with the previous models but this time with 'GaussianNB', then we execute the command for calculating the voting found in the Listing 22.

The output of this redesign of the models is the following accuracy table ordered by F1 score of macro average:

	Accuracy	F1-score (class 1)	F1-score (class 0)	F1-score (macro avg)
RF-best	0.852699	0.570667	0.911099	0.740883
voting_soft	0.850869	0.548476	0.910685	0.729581
voting_hard	0.844465	0.519774	0.907205	0.71349
DT-best	0.84721	0.476489	0.910552	0.69352
GaussianNB-default	0.763038	0.506667	0.84407	0.675368

Table 5: Comparison of best Ensemble methods with voting classifier.

As a result, we check our metrics over this model and obtain that our results are very similar to the validation ones. This is our final output information with the prediction done with the test dataset:

```
                precision    recall  f1-score   support

         0.0       0.90      0.89      0.89      1775
         1.0       0.48      0.48      0.48       356

    accuracy                           0.82      2131
   macro avg       0.69      0.69      0.69      2131
weighted avg       0.83      0.82      0.83      2131

predicted 0.0 1.0
target
0.0 1586 189
1.0 184 172
```

The precision shows that out of all instances predicted as matches the 48% of them were actually match. The recall shows that the model is able to capture only a 48% of the actual matches present in the dataset. The F1-score for the positive class ('yes' to match) achieves a reasonable balance between precision and recall of 48%. The accuracy value is good since it is an 78%, meaning that it correctly predicts the outcome ('match' or not) for approximately 78% of the instances. Looking at the confusion matrix, we can observe that out of the total instances predicted as not a match (0.0), 1586 instances were actually not a match, while 189 instances were predicted as not a match but were actually a match. Similarly, out of the total instances predicted as a match (1.0), 184 instances were actually not a match, while 172 instances were correctly predicted as a match.

The accuracy of the validation and test sets is quite similar, so we can say that the training of the model is succesful.

6 Conclusion

In summary, through this analysis where we have pre-processed the dataset, applied linear classification to the data with different methods, proceeded with feature selection to observe which are the decision variables over the partner selection and applied ensemble methods to obtain a prediction of this veredict.

The most relevant features make sense with their meaning and what do they imply to the dataset. First, the most valuable variable for the optimum accuracy is if the couple interviewed likes each other; which is considered a most adequate factor when choosing to say 'match' or not in a 4 minute date. Second and third, the variables are attractive and funny which represent a rating by partner (about me) at a night of event on attractiveness and funny; which falls accordingly to the 'like' variable and tells that partners give high importance to having fun. The following variables are all related to the likeness, attractiveness and funny; there is also greatly valued the shared interests aspect. Also notice that the personal qualities such as ambition, intelligence and sincere are valued; the personal interests such as theater, movies and shopping; and the age and race of the partner are also taken into account when deciding for 'match'.

In conclusion, the best method for making a prediction about the decision outcome for choosing a partner in a 4 minute date is the Random Forest method, which makes sense because it is an aggregation method that uses multiple decision trees to obtain a robust prediction. Furthermore, the features that have a higher weight for making this decision are the ones related to likeness, attractiveness and personal qualities of funny, ambition and intelligence.

7 References

- Ray Fisman & Sheena Iyengar, Speed Dating. 2004. University of Columbia Business School.
- Ray Fisman, Sheena Iyengar, Emir Kamenica & Itamar Simonson, "Gender Differences in Mate Selection: Evidence From a Speed Dating Experiment". May of 2006. The Quarterly Journal of Economics.

8 Appendix

Here is the list that contains all the features in the dataset:

```
* gender: Gender of self
* age: Age of self
* age_o: Age of partner
* d_age: Difference in age
* race: Race of self
* race_o: Race of partner
* samerace: Whether the two persons have the same race or not.
* importance_same_race: How important is it that partner is of same race?
* importance_same_religion: How important is it that partner has same
   religion?
* field: Field of study
* pref_o_attractive: How important does partner rate attractiveness
* pref_o_sinsere: How important does partner rate sincerity
* pref_o_intelligence: How important does partner rate intelligence
* pref_o_funny: How important does partner rate being funny
* pref_o_ambitious: How important does partner rate ambition
* pref_o_shared_interests: How important does partner rate having shared
   interests
* attractive_o: Rating by partner (about me) at night of event on
   attractiveness
* sincere_o: Rating by partner (about me) at night of event on sincerity
* intelligence_o: Rating by partner (about me) at night of event on
   intelligence
* funny_o: Rating by partner (about me) at night of event on being funny
* ambitous_o: Rating by partner (about me) at night of event on being
   ambitious
* shared_interests_o: Rating by partner (about me) at night of event on
   shared interest
* attractive_important: What do you look for in a partner - attractiveness
* sincere_important: What do you look for in a partner - sincerity
* intellicence_important: What do you look for in a partner - intelligence
* funny_important: What do you look for in a partner - being funny
* ambtition_important: What do you look for in a partner - ambition
* shared_interests_important: What do you look for in a partner - shared
   interests
* attractive: Rate yourself - attractiveness
* sincere: Rate yourself - sincerity
* intelligence: Rate yourself - intelligence
* funny: Rate yourself - being funny
* ambition: Rate yourself - ambition
* attractive_partner: Rate your partner - attractiveness
* sincere_partner: Rate your partner - sincerity
* intelligence_partner: Rate your partner - intelligence
* funny_partner: Rate your partner - being funny
* ambition_partner: Rate your partner - ambition
* shared_interests_partner: Rate your partner - shared interests
* sports: Your own interests [1-10]
```

```
 * tvsports
 * exercise
 * dining
 * museums
 * art
 * hiking
 * gaming
 * clubbing
 * reading
 * tv
 * theater
 * movies
 * concerts
 * music
 * shopping
 * yoga
 * interests_correlate: Correlation between participants and partners
     ratings of interests.
 * expected_happy_with_sd_people: How happy do you expect to be with the
     people you meet during the speed-dating event?
 * expected_num_interested_in_me: Out of the 20 people you will meet, how
     many do you expect will be interested in dating you?
 * expected_num_matches: How many matches do you expect to get?
 * like: Did you like your partner?
 * guess_prob_liked: How likely do you think it is that your partner likes
     you?
 * met: Have you met your partner before?
 * decision: Decision at night of event.
 * decision_o: Decision of partner at night of event.
 * match: Match (yes/no)
```

Listing 8: Complete list of features

Here is the code for the resampling of the original dataset:

```python
# Partition according to target feature 'match'
X = SD
y = SD["match"]
y=y.astype('float64')
# Split into a train set and a test set
SD_train, SD_test, match_train, match_test = train_test_split(X, y,
    test_size=0.33, random_state=42)
X1_train, X_val, y1_train, y_val = train_test_split(SD_train, match_train,
    test_size=0.25, stratify=match_train, random_state=42)
```

Listing 9: Resampling code

Here is the code that contains the code for One-Hot Encoding method:

```python
categorical_features = SD.dtypes[SD.dtypes == 'object'].index.values.tolist()
SD_categorical = SD[categorical_features]

# Preprocess categorical variables using one-hot encoding
```

```python
categorical_features.remove('match')
preprocessor = ColumnTransformer(
    transformers=[
        ('cat', OneHotEncoder(), categorical_features)
    ], remainder='passthrough'
)

# Partition according to target variable
X = SD_categorical.drop("match", axis=1)
y = SD["match"]

# Performing one-hot-encoding
pipeline = Pipeline(steps=[('preprocessor', preprocessor)])
out = pipeline.fit_transform(X, y)

# SD after one-hot-encoding
SD_one_hot_encoding = pd.DataFrame(out.toarray())
```

Listing 10: One-hot-encoding

Here is the code that changest the header after one-hot encoding:

```python
from sqlalchemy.sql.expression import all_

# for every categorical feature
all_columns = []
for feature in categorical_features:
  # compute the total of categories
  total = len(SD[feature].unique());
  all_columns = all_columns + [feature + '_' + str(i) for i in range(total)]

# update dataframe's header
SD_one_hot_encoding.columns = all_columns
```

Listing 11: Header after one-hot-encoding

Here is the code that scales the training and test set separately:

```python
def preprocessing(X, y, scaler=None):
    # We scale all the columns
    if scaler is None:
        # We only want the scaler to fit the train data
        scaler = MinMaxScaler()
        # Normalize the training data using Min-Max scaling
        X.loc[:,numerical_features] =
            scaler.fit_transform(X[numerical_features])
    else:
        X.loc[:,numerical_features] = scaler.transform(X[numerical_features])
    return X, y, scaler

X_train, y_train, scaler = preprocessing(SD_train,match_train)
X_test, y_test, _ = preprocessing(SD_test,match_test,scaler)
```

Listing 12: Min-max Scaling

Here is the code for LDA method for preprocessed data:

```
lda_model = LinearDiscriminantAnalysis()
lda_model = lda_model.fit(X_train, y_train)

print('Priors:', lda_model.priors_)
print('Means:\n')
means =pd.DataFrame(lda_model.means_)
means.columns=SD_preprocessed.columns[0:]
means.index = lda_model.classes_
means
# Compute cross-validation metrics for LDA
cross_val_results = pd.DataFrame(cross_validate(lda_model , X_train,
    y_train, cv = 5, scoring = ['accuracy', 'f1_macro', 'precision_macro',
    'recall_macro'] ))
metrics.loc['LDA',:] = cross_val_results[['test_accuracy', 'test_f1_macro',
        'test_precision_macro', 'test_recall_macro']].mean().values
metrics
```

Listing 13: LDA with preprocessed dataset

Here is the code for KNN method:

```
knn = KNeighborsClassifier()
knn_cv = GridSearchCV(
    estimator=knn,
    param_grid={
        'n_neighbors': [1, 3, 5, 7, 10, 15, 20],
        'metric': ['euclidean', 'minkowski', 'manhattan']
    },
    scoring=['accuracy', 'f1_macro', 'precision_macro', 'recall_macro'],
    refit=False
)
knn_cv.fit(X_train, y_train)
results_cv = pd.DataFrame(knn_cv.cv_results_)
```

Listing 14: KNN with preprocessed dataset

Here is the code for Logistic regression method:

```
logreg = LogisticRegressionCV(Cs=20, random_state=1, cv = 10, scoring =
    'accuracy', multi_class='multinomial')
logreg.fit(X_train, y_train)
```

Listing 15: Logistic Regression with preprocessed dataset

Here is the code for the prediction and confusion matrix:

```
y_pred = logreg.fit(X_train, y_train).predict(X_test)
confusion(y_test, pd.Series(y_pred))
print(classification_report(y_test, y_pred))
```

Listing 16: Prediction

Here is the code for Lasso regression:

```python
# Create LASSO model with cross-validation
lasso_model = LassoCV(cv=5) # Removed (, normalize=True) clause

# Create a pipeline to fit the LASSO model
pipeline = Pipeline(steps=[('model', lasso_model)])

# Fit the LASSO model to the data
pipeline.fit(X_train, y_train)

# Get the LASSO coefficients
lasso_coefficients = pipeline.named_steps['model'].coef_

# Get the feature names (including one-hot encoded features)
feature_names = SD_preprocessed.columns

# Print the LASSO coefficients along with the feature names
for feature, coef in zip(feature_names, lasso_coefficients):
    print(f"{feature}: {coef}")

# Identify the important features (non-zero coefficients)
important_features = [feature for feature, coef in zip(feature_names,
    lasso_coefficients) if coef != 0]

# Print the important features
print("\nImportant features:")
print(important_features)
```

Listing 17: Lasso Regression

Here is the code for the decision tree method of Ensembles:

```python
model_tree = DecisionTreeClassifier().fit(X_train, y_train)
y_pred = model_tree.predict(X_val)
results.loc['DT-default',:] = compute_metrics(y_val, y_pred)
confusion(y_val, y_pred)
print(classification_report(y_val,
                            y_pred,
                            target_names=['no', 'yes'],))
results
```

Listing 18: Decision tree code

Here is the code for Random forest:

```python
model_rf1 = RandomForestClassifier(oob_score=True).fit(X_train, y_train)
pred = model_rf1.predict(X_train)
confusion(y_train,pred)
print(classification_report(y_train,
                            pred,
                            target_names=['no', 'yes'],))
print('OOB accuracy=', model_rf1.oob_score_)
```

Listing 19: Random Forest Method

Here is the code for real validation error for random forest:

```
y_pred = model_rf1.predict(X_val)
print('Validation Accuracy:{}'.format(model_rf1.score(X_val,y_val)))
results.loc['RF-default',:] = compute_metrics(y_val,y_pred)
confusion(y_val,y_pred)
```

Listing 20: Real validation error for random forest

Here is the code for generating a graphic of feature importance:

```
var_imp = pd.DataFrame({'importance':
    rf_model_tuned.feature_importances_},index=X_train.columns)
var_imp.sort_values(by='importance').loc[var_imp['importance'] >
    0.005].plot.barh(figsize=(8,8),legend=True);
```

Listing 21: Feature importance graphic

Here is the code for the voting classifier:

```
voting_hard = VotingClassifier([('dt', model_tree), ('rf',
    model_rf1),('gnb', gauss_nb)])
voting_hard.fit(X_train, y_train)
y_pred = voting_hard.predict(X_val)
results.loc['voting_hard', :] = compute_metrics(y_val, y_pred)
results.loc[['DT-best','GaussianNB-default','RF-best','voting_hard'],:].
sort_values(by='F1-score (class 1)',ascending=False)
```

Listing 22: Calculation of voting classifier