Machine Learning Project

Mohammad Zain Abbas
Zyrako Musaj

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

June, 2022

You can find more detail on our GitHub repository

1. Introduction

1.1 Problem definition

In this project, we have investigated whether it is possible to **predict if a partner will match with their date** based on **dating preferences**, **attribute ratings** and **background information**. The problem can be framed as a supervised, binary classification problem where the model predicts if a partner has accepted or rejected their date.

1.2 Motivation

In today's busy world, finding and dating a romantic partner seems more time consuming than ever. As a result, many people have turned to speed dating as a solution that allows one to meet and interact with a large number of potential partners in a short amount of time. In this report, we want to explore what people are looking for in their speed dating matches, what it takes to become successful in getting approvals from a potential partner, if there exist any gender differences, and if any other factors (such as the order you met your partner) influence peoples' decisions. Finally, we'd like to determine if people really know what they want by comparing their self-reported answers to what actually influences peoples' decisions.

1.3 Dataset

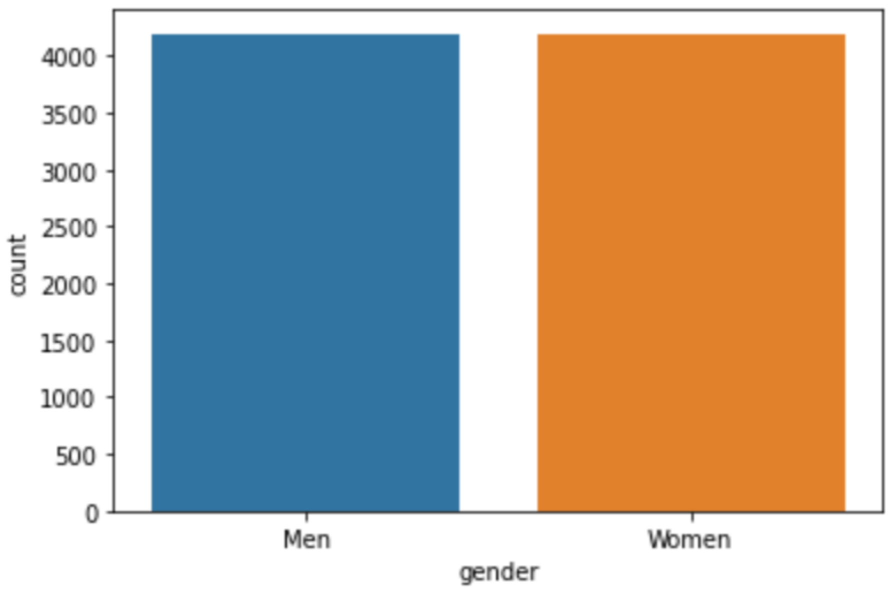
The data set that we have explored in the project is named Speed Dating Experiment, and can be found on Kaggle.com. It was compiled by professors *Ray Fisman* and *Sheena Iyengar* from **Columbia Business School**, originally used for their paper Gender Differences in Mate Selection: Evidence From a Speed Dating Experiment. And later for Racial Preferences in Dating with more detailed analysis. It was generated from a series of experimental speed dating events from 2002 to 2004 and includes data related to demographics, dating habits, lifestyle information, an attribute evaluation questionnaire taken when the participants sign up, and each participant's ratings for others during the 4 minute interactions. Finally, individuals were asked if they would like a second date with their partners and rated again on similar questions after the event, when matches have met with each other and dated for several times.

2. Exploratory data analysis (EDA)

Before starting working on the dataset and modeling etc, it's crucial to build a solid understanding of the dataset first. You can find detailed EDA here on our github repo. List below are some key highlights of our EDA:

2.1 Gender based Analysis

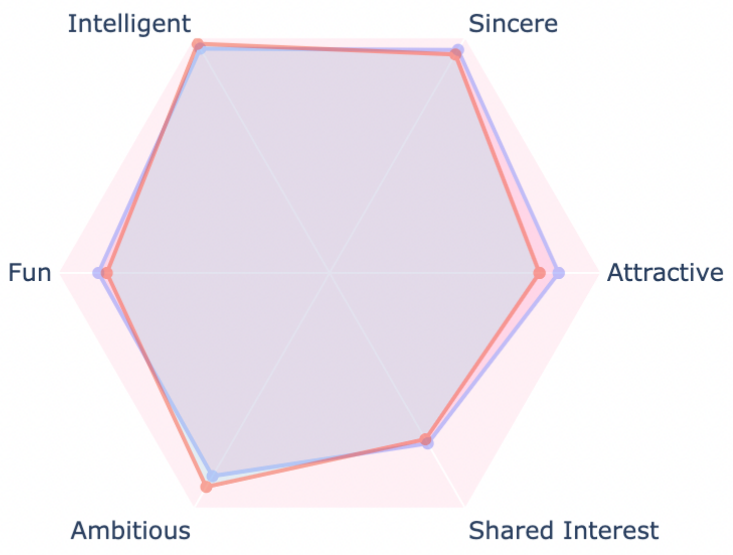
Since our dataset included participants from both genders, we first looked at the ratio between male and female participants.



Ratio b/w male and female participants

It's clear that we have a pretty balanced dataset in terms of the participant's gender.

2.1.1 Which attributes does each gender look for ?

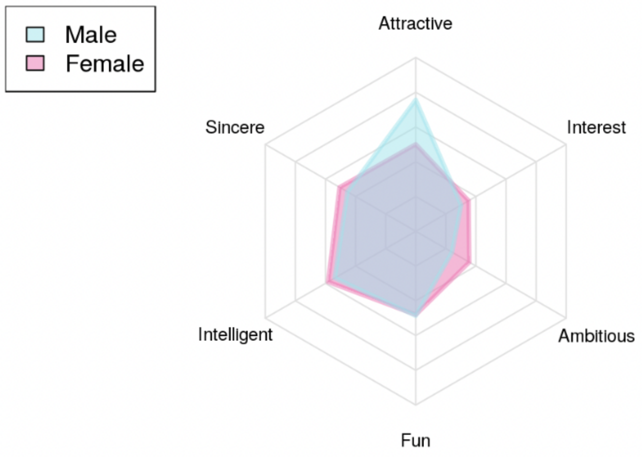


Attributed each gender look for

Both genders are generally evenly distributed across all of the attributes. However, men generally look for attractive women and Women are looking for a well-rounded male.

2.1.2 What are participants looking for in their matches ?

We were interested to see what do the participants in these speed dating events look for in the opposite sex, and if there exist a difference for male and female participants. At this point in time, the participants have just signed up for the event and have not met anyone.

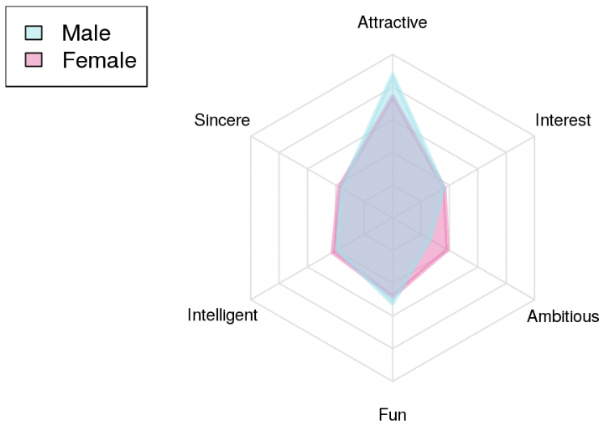


What participants look for in their match

Men are looking for attractive women, and are less concerned with a woman's ambition and shared interests. On the other hand, women are looking for a well-rounded male and value intelligence in a man.

2.1.3 What paricipants think their same-sex peers are looking for ?

Next, we want to explore what people think men/women of their same sex is looking for, learn how the participants view their competitors, and determine if they separate their own views from the majority.

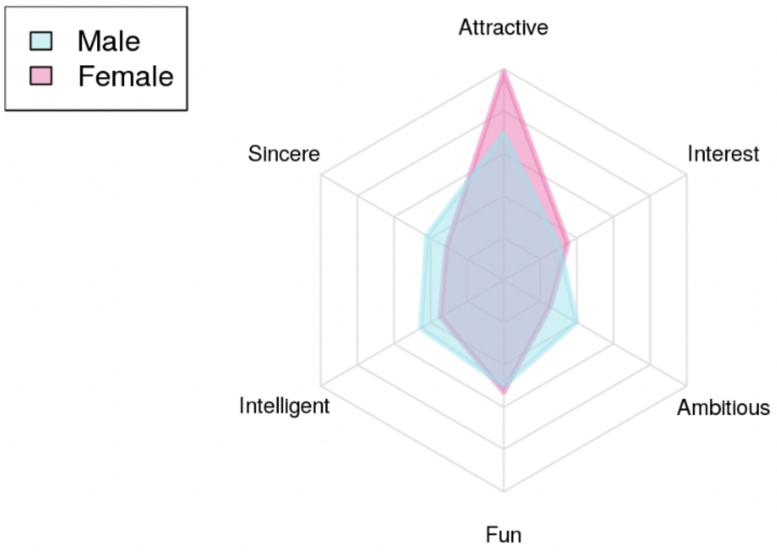


What paricipants think their same-sex peers are looking for ?

Both men and women think that their competitors are looking for attractive counterparts, and what women say they are looking for is drastically different from what other women think their peers value.

2.1.4 What do participants think the opposite sex is looking for ?

Finally, we analyze what participants think their opposite sex is looking for. We will be able to see if there are any differences in the expectations of men and women with regards to the speed dating event.

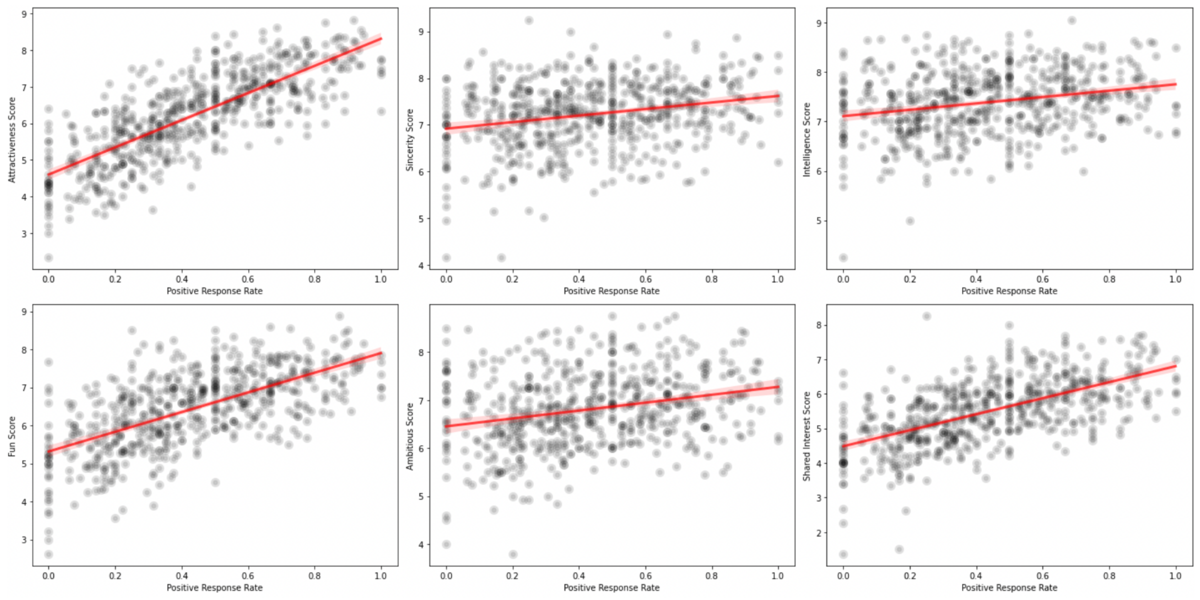


What do participants think the opposite sex is looking for ?

Both men and women can predict what the opposite sex are looking for in their partners to a certain degree.

2.2 Attribute based analysis

We calculated how many positive responses can a person get from the opposite sex. For example, if a man met with 10 women and 6 of them indicated that they would like to meet him again, his positive response rate would be 60%. On the y-axis, we averaged the scores in the 6 attributes that the man received from the 10 women.



Influence of attributes to matchmaking (decision)

3. Preprocessing

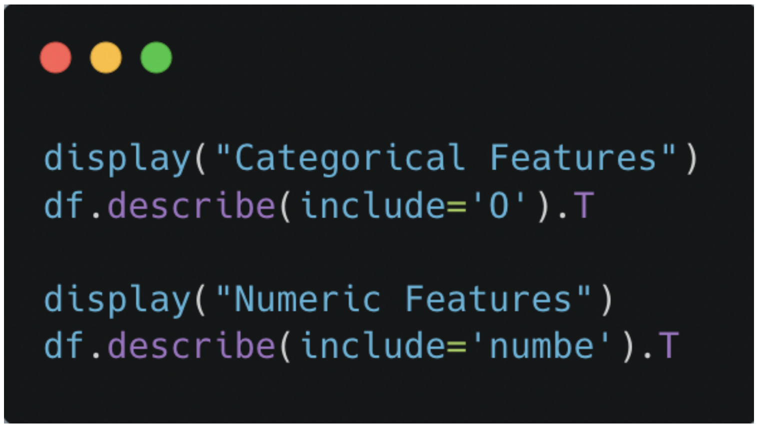
Data preprocessing is the first machine learning step in which we transform raw data obtained from various sources into a usable format to implement accurate machine learning models.

In a real-world data science project, data preprocessing is one of the most important things, and it is one of the common factors of success of a model, i.e: if there is correct data preprocessing and feature engineering, that model is more likely to produce noticeably better results as compared to a model for which data is not well preprocessed.

You can find detailed preprocessing here on our github repo. List below are some key highlights of our preprocessing:

3.1 Basic data checking

Generally, don't tend to care about data types until we get an error or some unexpected results. However, it's better to check that data types are correctly loaded before you start your analysis. You can read more about data types in pandas here



```
display("Categorical Features")
df.describe(include='O').T

display("Numeric Features")
df.describe(include='numbe').T
```

First we looked at the data types and noticed we have wrong data types for some features. So, we cast them to correct data types and verified their types again.

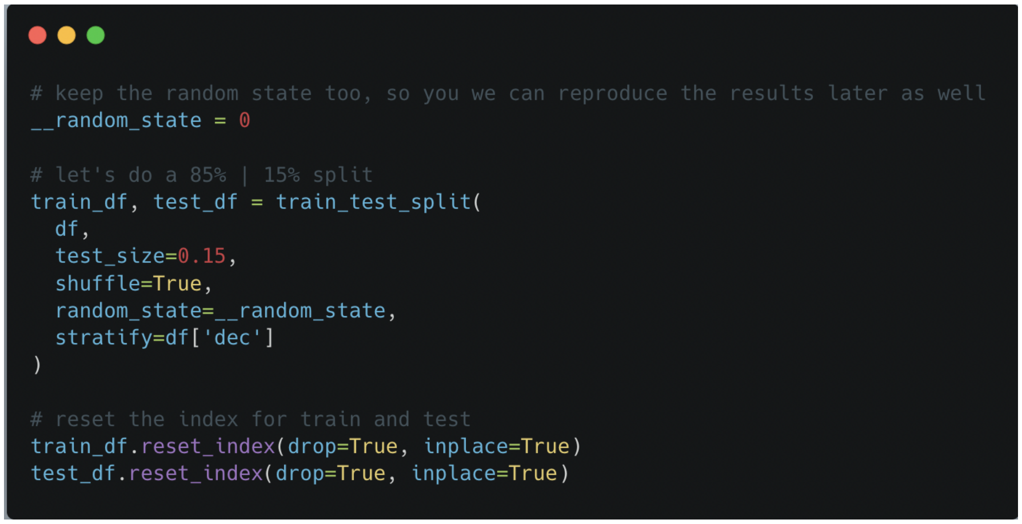
3.2 Column renaming and dropping irrelevant columns

We didn't need to make changes to the column names. However, we had to drop most of the columns which were not worth looking into. We have referred to the original paper about data acquisition and referred to the document which provided key mapping information, to get the idea about which features to keep and which ones to drop.

Fun fact: we also checked the memory being used by the dataframe. And we noticed a change when we changed the data types for some columns.

3.3 Split your dataset into train and test datasets

We split our dataset into two parts: *train* & *test* datasets. We will do all the processing on the *train* dataset. The *test* dataset will remain unknown to us. And we will use it only for testing analysis. This is to simulate the real world scenario in which we don't know the data which would be run on our model (after deploying)

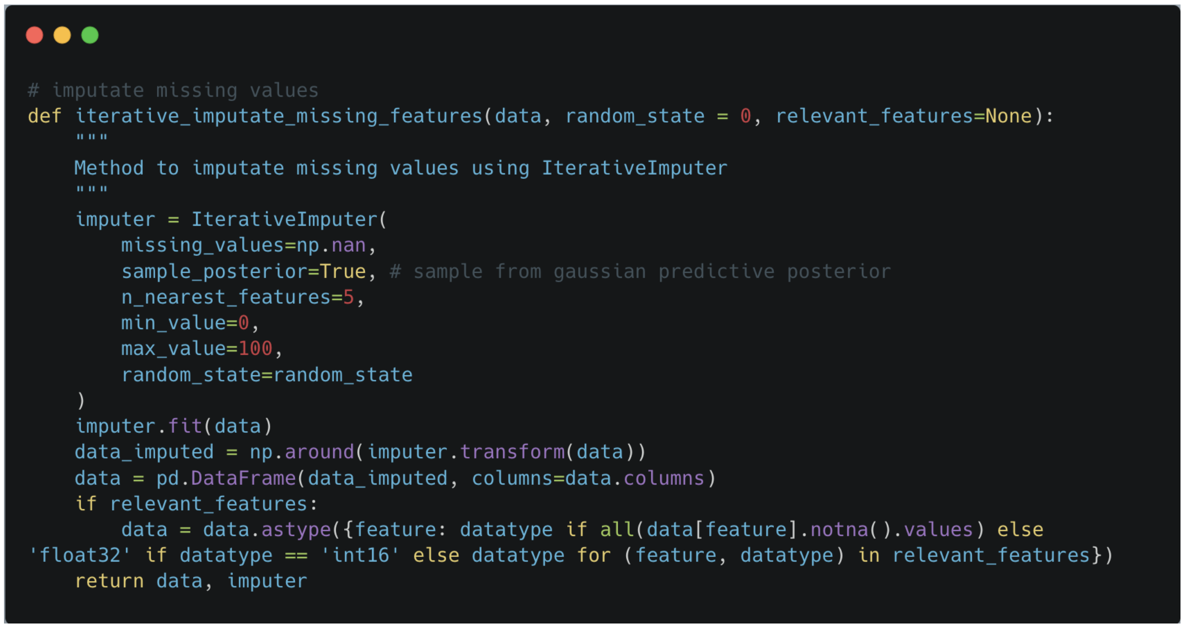


```python
# keep the random state too, so you we can reproduce the results later as well
__random_state = 0

# let's do a 85% | 15% split
train_df, test_df = train_test_split(
  df,
  test_size=0.15,
  shuffle=True,
  random_state=__random_state,
  stratify=df['dec']
)

# reset the index for train and test
train_df.reset_index(drop=True, inplace=True)
test_df.reset_index(drop=True, inplace=True)
```

Split dataset into train and test dataset

3.4 Checking for missing data

We checked for missing data and imputed or removed it. It is really important to deal with all the missing data to get better EDA and less incorrect results during model training.



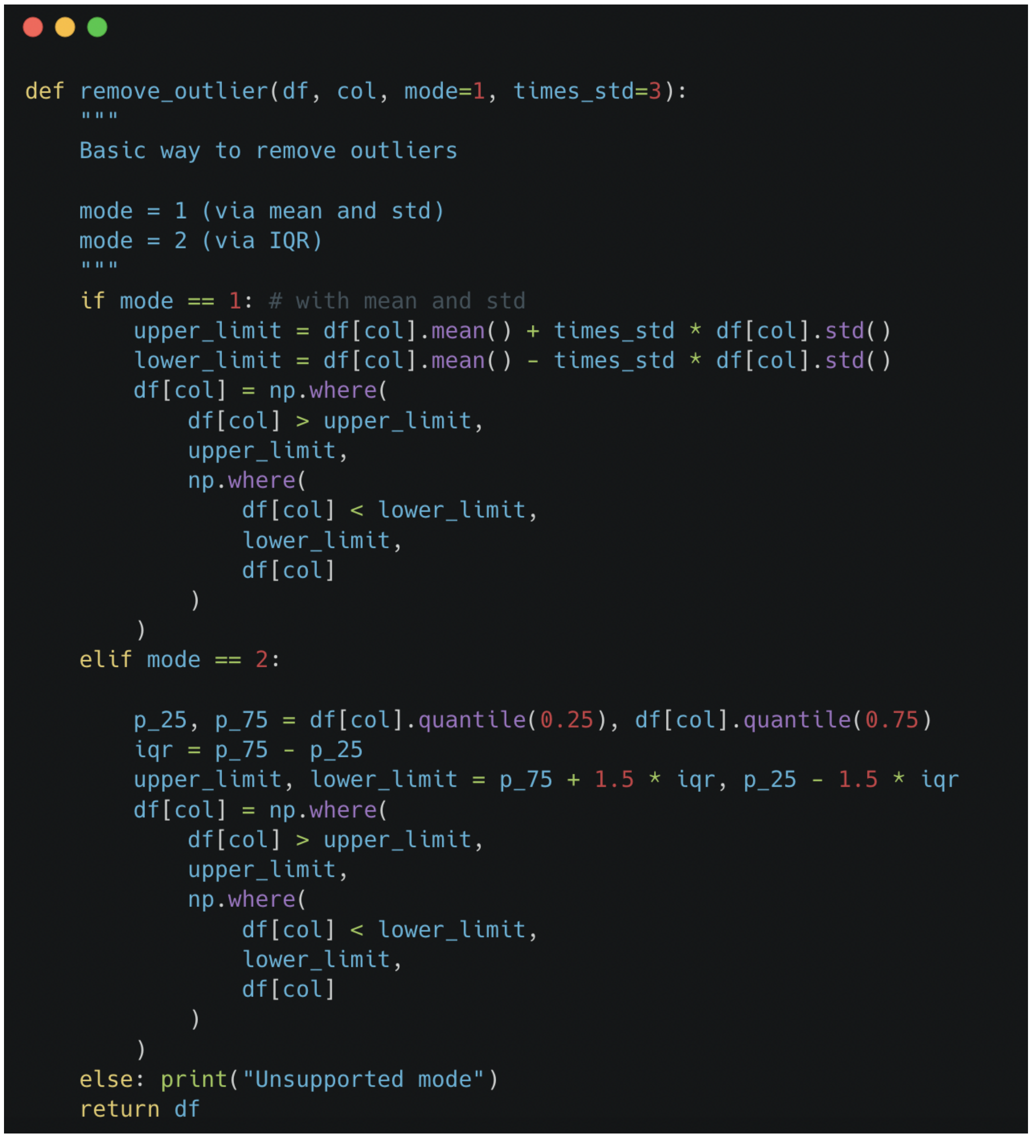
```python
# imputate missing values
def iterative_imputate_missing_features(data, random_state = 0, relevant_features=None):
    """
    Method to imputate missing values using IterativeImputer
    """
    imputer = IterativeImputer(
        missing_values=np.nan,
        sample_posterior=True, # sample from gaussian predictive posterior
        n_nearest_features=5,
        min_value=0,
        max_value=100,
        random_state=random_state
    )
    imputer.fit(data)
    data_imputed = np.around(imputer.transform(data))
    data = pd.DataFrame(data_imputed, columns=data.columns)
    if relevant_features:
        data = data.astype({feature: datatype if all(data[feature].notna().values) else
'float32' if datatype == 'int16' else datatype for (feature, datatype) in relevant_features})
    return data, imputer
```

Imputation method

We used IterativeImputer to impute the missing data by sampling from gaussian predictive posterior. And then later saved the imputer as a pickle file (to preprocess our test dataset)

3.5 Outlier detection and removal

We used Box-plots to visualize the outliers. And afterwards, we had the option to use various methods to remove the outliers: like z-score, model based (one class SVM and density based algorithms etc). We have tried to keep it simple and used basic statistics (mean and std) to eliminate the outliers. To further improve our dataset, we used IQRs to remove the outliers.



```python
def remove_outlier(df, col, mode=1, times_std=3):
    """
    Basic way to remove outliers

    mode = 1 (via mean and std)
    mode = 2 (via IQR)
    """
    if mode == 1: # with mean and std
        upper_limit = df[col].mean() + times_std * df[col].std()
        lower_limit = df[col].mean() - times_std * df[col].std()
        df[col] = np.where(
            df[col] > upper_limit,
            upper_limit,
            np.where(
                df[col] < lower_limit,
                lower_limit,
                df[col]
            )
        )
    elif mode == 2:

        p_25, p_75 = df[col].quantile(0.25), df[col].quantile(0.75)
        iqr = p_75 - p_25
        upper_limit, lower_limit = p_75 + 1.5 * iqr, p_25 - 1.5 * iqr
        df[col] = np.where(
            df[col] > upper_limit,
            upper_limit,
            np.where(
                df[col] < lower_limit,
                lower_limit,
                df[col]
            )
        )
    else: print("Unsupported mode")
    return df
```

Method for outlier removal using basic statistics

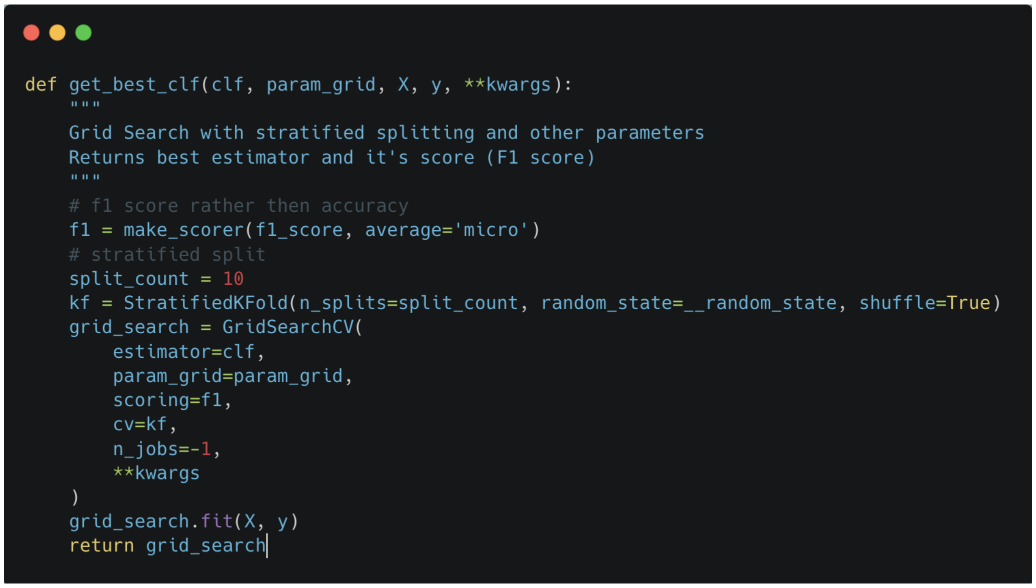
3.6 Feature engineering

We first encoded nominal features using one-hot encoding. Then we calculated the average attribute ratings for each subject. Then we inserted average attribute ratings to the dataframe. Afterwards, we calculated the difference between subject and partner's ages. Also, the difference between subject's attribute ratings and partner's attributes ratings.
We figured that we needed scaling too. So, we scaled normal features to zero mean and unit variance by z-scoring. We also noticed that there were some irrelevant features, which we got rid of at the end.

After all these transformations, we saved the processed data into a csv file (in order to be used by the next step i.e: modeling)

4. Modeling

For modeling, we decided to use two main classes of classifiers/estimators, i.e: *Baseline models* and *Ensemble models*. We used GridSearchCV to exhaustively search over specified parameter values for each estimator. As a cross-validation splitting strategy, we used stratified k-fold splitting with 10 splits, to ensure uniform and homogeneous results.



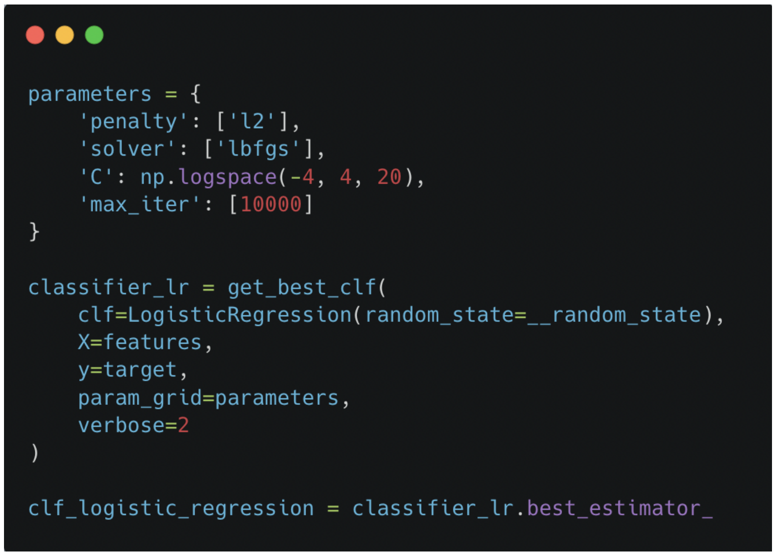
```python
def get_best_clf(clf, param_grid, X, y, **kwargs):
    """
    Grid Search with stratified splitting and other parameters
    Returns best estimator and it's score (F1 score)
    """
    # f1 score rather then accuracy
    f1 = make_scorer(f1_score, average='micro')
    # stratified split
    split_count = 10
    kf = StratifiedKFold(n_splits=split_count, random_state=__random_state, shuffle=True)
    grid_search = GridSearchCV(
        estimator=clf,
        param_grid=param_grid,
        scoring=f1,
        cv=kf,
        n_jobs=-1,
        **kwargs
    )
    grid_search.fit(X, y)
    return grid_search
```

Method for exhaustively search over parameters with stratified splitting and with F1 score as base metric

4.1 Baseline Models

4.1.1 Logistic Regression

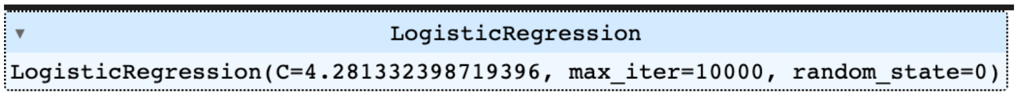
First, we decided to apply a linear model: Logistic regression (aka logit or MaxEnt) classifier with following grid parameters (which will be used in grid search to determine the best classifier/estimator).



```python
parameters = {
    'penalty': ['l2'],
    'solver': ['lbfgs'],
    'C': np.logspace(-4, 4, 20),
    'max_iter': [10000]
}

classifier_lr = get_best_clf(
    clf=LogisticRegression(random_state=__random_state),
    X=features,
    y=target,
    param_grid=parameters,
    verbose=2
)

clf_logistic_regression = classifier_lr.best_estimator_
```

Logistic regression's params and getting best estimator after grid search

We got the following regression model as our best estimator:

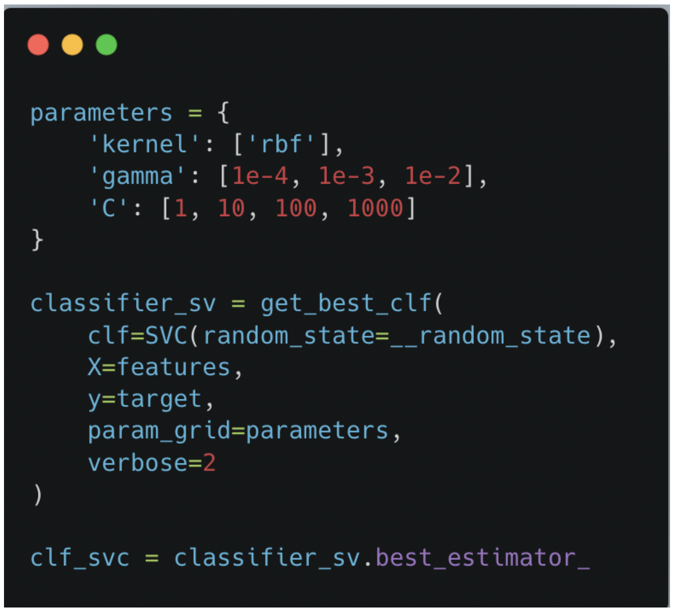


```
▾ LogisticRegression
LogisticRegression(C=4.281332398719396, max_iter=10000, random_state=0)
```

Best logistic regression's estimator (after grid search)

4.1.2 SVC

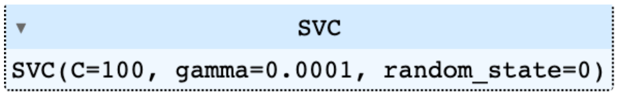
We applied Linear Support Vector Classifier (SVC) with following params:



```
parameters = {
    'kernel': ['rbf'],
    'gamma': [1e-4, 1e-3, 1e-2],
    'C': [1, 10, 100, 1000]
}

classifier_sv = get_best_clf(
    clf=SVC(random_state=__random_state),
    X=features,
    y=target,
    param_grid=parameters,
    verbose=2
)

clf_svc = classifier_sv.best_estimator_
```

Linear support vector classifier's params and getting best estimator after grid search

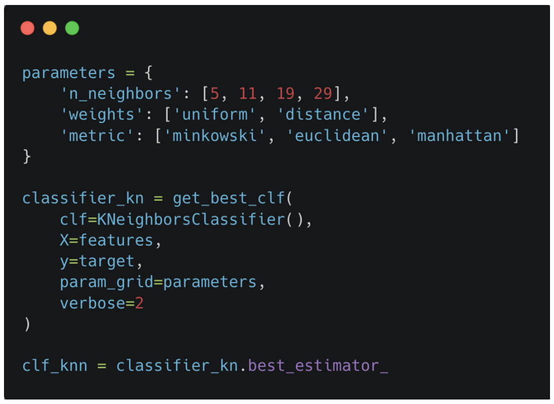
We got the following regression model as our best estimator:



```
▾ SVC
SVC(C=100, gamma=0.0001, random_state=0)
```

4.1.3 KNN

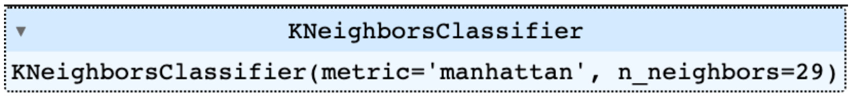
In the end, we decided to use KNN as well.



```
parameters = {
    'n_neighbors': [5, 11, 19, 29],
    'weights': ['uniform', 'distance'],
    'metric': ['minkowski', 'euclidean', 'manhattan']
}

classifier_kn = get_best_clf(
    clf=KNeighborsClassifier(),
    X=features,
    y=target,
    param_grid=parameters,
    verbose=2
)

clf_knn = classifier_kn.best_estimator_
```

KNN classifier's params and getting best estimator after grid search

We got the following regression model as our best estimator:

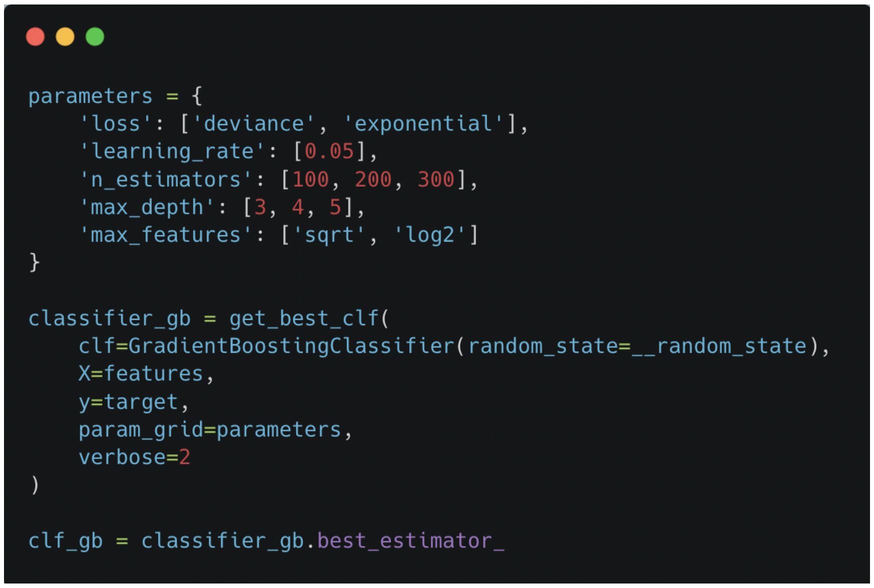


```
KNeighborsClassifier
KNeighborsClassifier(metric='manhattan', n_neighbors=29)
```

4.2 Ensemble Models

4.2.1 Gradient Boost

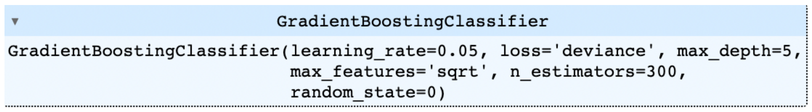
In ensemble models, we started with Gradient Boosting for classification, with below mentioned params:



```
parameters = {
    'loss': ['deviance', 'exponential'],
    'learning_rate': [0.05],
    'n_estimators': [100, 200, 300],
    'max_depth': [3, 4, 5],
    'max_features': ['sqrt', 'log2']
}

classifier_gb = get_best_clf(
    clf=GradientBoostingClassifier(random_state=__random_state),
    X=features,
    y=target,
    param_grid=parameters,
    verbose=2
)

clf_gb = classifier_gb.best_estimator_
```

Gradient Boost classifier's params and getting best estimator after grid search

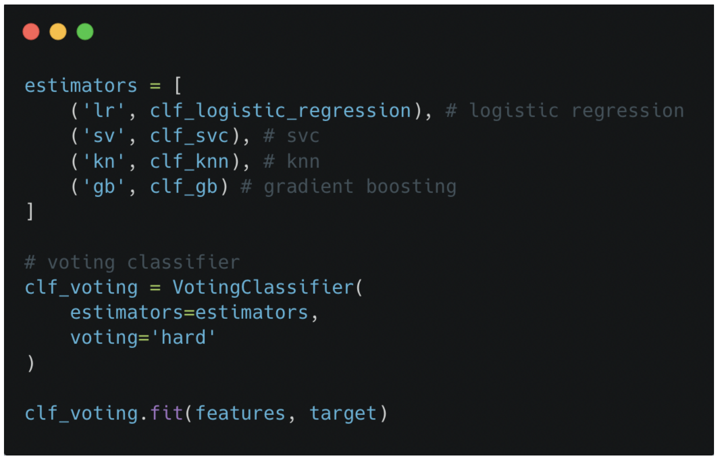
We got the following regression model as our best estimator:



```
GradientBoostingClassifier
GradientBoostingClassifier(learning_rate=0.05, loss='deviance', max_depth=5,
                           max_features='sqrt', n_estimators=300,
                           random_state=0)
```

4.2.2 Voting Classifier

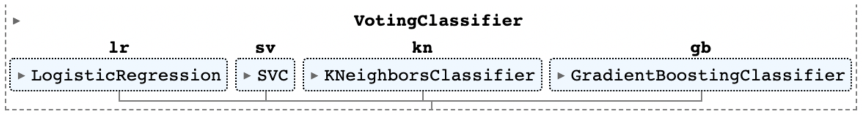
We then applied soft voting/majority rule classifier for unfitted estimators via VotingClassifier



```
estimators = [
    ('lr', clf_logistic_regression), # logistic regression
    ('sv', clf_svc), # svc
    ('kn', clf_knn), # knn
    ('gb', clf_gb) # gradient boosting
]

# voting classifier
clf_voting = VotingClassifier(
    estimators=estimators,
    voting='hard'
)

clf_voting.fit(features, target)
```

Voting classifier's params and getting best estimator after grid search

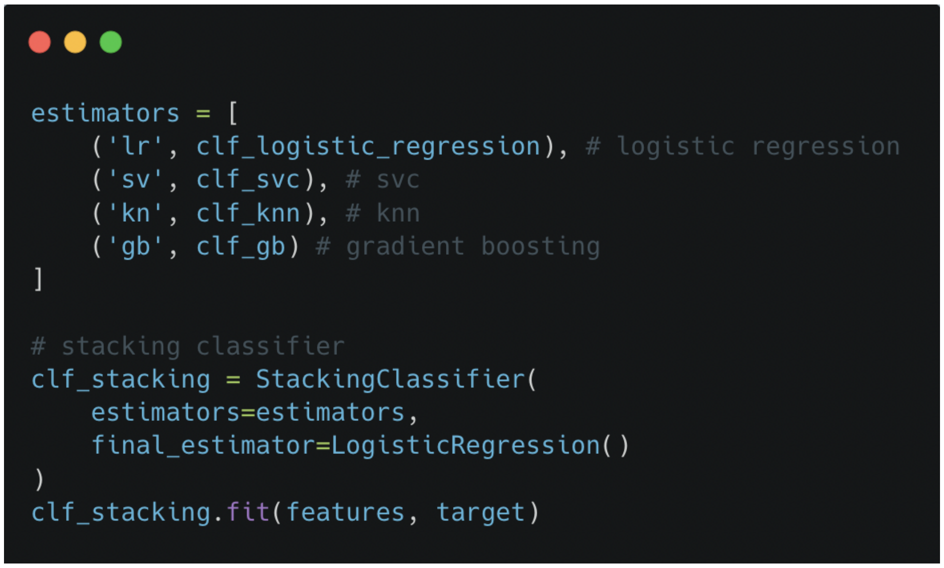
We got the following Voting classifier model:



Voting classifier

4.2.3 Stacking Classifier

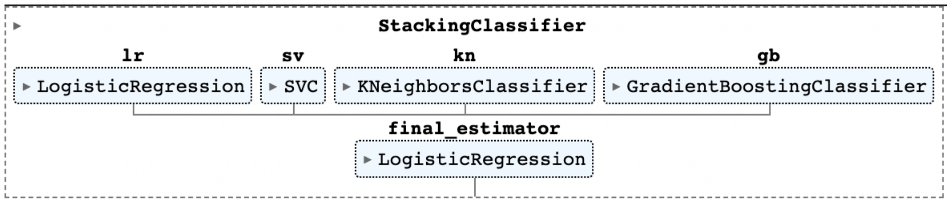
At the end, we applied a stack of classifiers/estimators with a final classifier. We picked LogisticRegression as the final estimator.



```
estimators = [
    ('lr', clf_logistic_regression), # logistic regression
    ('sv', clf_svc), # svc
    ('kn', clf_knn), # knn
    ('gb', clf_gb) # gradient boosting
]

# stacking classifier
clf_stacking = StackingClassifier(
    estimators=estimators,
    final_estimator=LogisticRegression()
)
clf_stacking.fit(features, target)
```

Stacking Classifier's params and getting best estimator after grid search

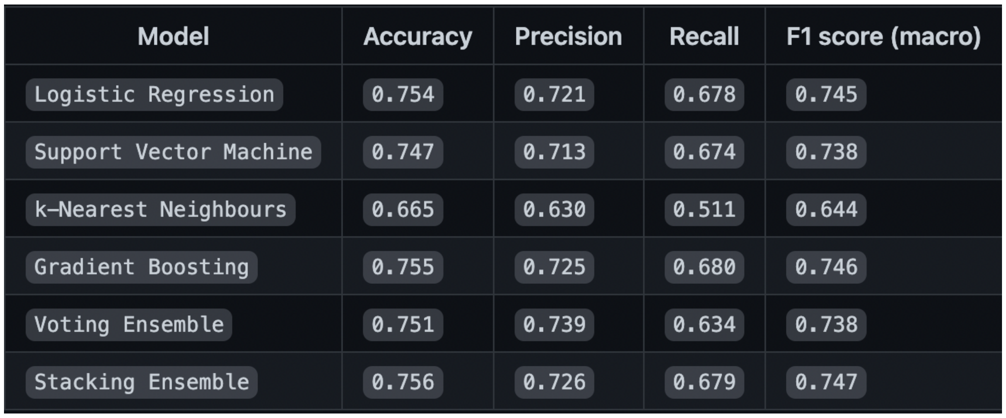
We got the following Stacking classifier model:



Stacking classifier

5. Test Analysis

At the end of a typical machine learning project, we analyze our test dataset with the pipelines we had developed in previous stages. We used the same imputation model (as discussed in the *preprocessing* section).



Model	Accuracy	Precision	Recall	F1 score (macro)
`Logistic Regression`	`0.754`	`0.721`	`0.678`	`0.745`
`Support Vector Machine`	`0.747`	`0.713`	`0.674`	`0.738`
`k-Nearest Neighbours`	`0.665`	`0.630`	`0.511`	`0.644`
`Gradient Boosting`	`0.755`	`0.725`	`0.680`	`0.746`
`Voting Ensemble`	`0.751`	`0.739`	`0.634`	`0.738`
`Stacking Ensemble`	`0.756`	`0.726`	`0.679`	`0.747`

Comparison between different techniques

5.1 Final model

After analyzing the score, the stacking ensemble model is performing better than the rest, with F1 score ~0.747 and accuracy of ~0.756.

5.2 Scientific and personal conclusions

Based on correlation analysis and weights in the linear model, we have concluded that:

Attractiveness, ***shared interests*** and being ***fun*** were the most significant factors in a partner's decision. *Medical students* had the highest probability of being matched while *psychologists* and *academics* had the lowest.

Further feature engineering such as introducing more interaction features and performing more thorough feature selection could improve the classifier performance.

NOTE: Attached notebooks may give some issues. (I wasn't able to solve them in due time owing to bad health). So kindly refer to the notebooks on github to see the latest version of code. You can refer here for more detail.

References:

1. Speed Dating Experiment - Kaggle
2. Gender Differences in Mate Selection: Evidence From a Speed Dating Experiment
3. Racial Preferences in Dating

All other resources or materials are mentioned in the notebooks (at the end of each notebook).