Affinity Predictor

Arduini Luca

List of contents

Introduction

In today's digital age, the way people build interpersonal relationships has changed dramatically. Online dating apps like Tinder, Bumble, and Hinge have become widespread tools, especially among younger generations, for meeting new people and finding potential partners.

This mechanism of almost instant evaluation, although technologically modern, echoes traditional decision-making dynamics, such as those seen in speed dating events. This project fits right into that context, using a rich dataset based on speed dating experiments as a testing ground for developing an intelligent recommendation system.

The **final goal** of the project is to implement a Machine Learning classifier for a hypothetical dating app. The aim is to create a smart filter that improves the user experience by suggesting profiles with a high probability of mutual compatibility. To achieve this, the model will be trained to predict the target variable decision (whether a person would like to meet someone in real life), using all the available information in the dataset: from users' demographic characteristics to their interests and stated preferences.

In this way, the goal is to go beyond basic manual filters (like age range or distance) and offer a more personalized and relevant service. The algorithm would avoid showing users profiles they are unlikely to be interested in, based on the model's predictions, increasing both the efficiency and quality of suggested matches.

The code is available on GitHub at: github.com/LucaArduini/AffinityPredictor

Dataset

The dataset used in this Machine Learning project comes from an experimental study conducted between 2002 and 2004 on speed dating events. The data was collected from graduate and professional students at Columbia University, who took part in structured speed dating sessions.

The experimental design required each participant to go on a "first date" lasting four minutes with every other participant of the opposite sex. At the end of each date, participants had to fill out a scorecard with two main types of information:

1. **The Decision**: A binary choice ("Yes" or "No") about whether the participant would like to meet the person again. This variable is called `decision` in the dataset. There is also `decision_o`, which represents the other person's response toward the participant. If both decisions are positive, then the variable `match` will have a value of 1, indicating that both people want to go on a second date.
2. **Partner Ratings**: A score for six key attributes to evaluate the person they just met: Attractiveness, Sincerity, Intelligence, Fun, Ambition, and Shared Interests.

In addition to the data collected during the dates, the dataset also includes information from questionnaires filled out by the participants before and after the event. These additional features cover a wide range of details such as demographic information, personal and dating habits, self-assessments of personality and traits, and opinions on what is important in a potential partner.

Data Description

The variables in the dataset can be grouped into three main thematic areas:

- **Demographic and Pair Data**: The questionnaires collected basic demographic information about each participant, such as gender, age (`age`), ethnicity (`race`), and field of study. Particularly interesting are the variables calculated for each specific pair, like age difference (`d_age`) and whether they belong to the same ethnic group (`samerace`). These features measure how similar the two people are and help analyze the level of compatibility between them.
- **Self-Perception and Stated Preferences**: A key aspect of the dataset is the ability to compare actual behavior with stated preferences. Participants not only rated themselves on the same six attributes, but also used a point system to indicate how important each attribute is to them in a potential partner. This makes it possible to explore the possible gap between ideal preferences and the real choices made during the event.
- **Post-Date Ratings**: Right after each "mini-date," participants gave immediate feedback on their partner using a scale from 1 to 10 for six key attributes: Attractiveness, Sincerity, Intelligence, Fun, Ambition, and Shared Interests. These ratings (`attr_o`, `sinc_o`, etc.) represent the initial impression and are among the most direct predictors of the final decision.

The final dataset consists of **8378 observations** and **123 features**, where each row represents a single meeting between two participants. There is a total of **1.78% missing values**, which, of course, do not affect the target variable.

Our Use of the Dataset

Since our idea is to implement a classifier for an online dating app, with the goal of showing users only profiles that are compatible with them, the original version of the dataset was not fully suitable. As mentioned earlier, it included many features based on post-date ratings of the partner, taken from surveys filled out the day after the event.

Obviously, in our use case, we don't have access to this kind of information. The only data available, and that we can base our work on, are the details we can ask users to provide during the registration process through a form. We certainly can't rely on information that would only be available after a meeting between potential partners.

So, the first step in our work was to remove from the dataset all the columns related to Post-Date Ratings. Here are some of the most obvious examples:

- `like`: Did you like your partner?
- `guess_prob_liked`: How likely do you think it is that your partner likes you?
- `attractive_partner`, `sincere_partner`, `intelligence_partner`, `funny_partner`, `ambition_partner`, `shared_interests_partner`: these variables contained the participant's evaluation of their partner on each attribute.
- Of course, we also removed the corresponding variables where the partner rated us on those same aspects.

We then removed the `decision_o` feature, since for each row where person A is the main participant and person B is the potential partner, the dataset also includes another row where the roles are reversed. This means we don't lose any valuable information by removing it.

Finally, we removed the `match` feature, because our goal is to learn as much as possible about how participant A evaluates a potential partner, not whether both sides agreed to meet again.

Preprocessing

In this chapter of the documentation, we will discuss the contents of the `preprocessing.ipynb` file. This notebook contains all the code used to preprocess the dataset and prepare it for the next phase of the project.

Dataset Acquisition

As expected, in this section we download the [dataset from the openml.org](openml.org) website, but not only that. Right after downloading it, we format the NaN values so that pandas can read them correctly.
Then, we remove all the columns that won't be available in our use case, as described in the previous chapter.

At this point, the dataset is ready to work on.

Data Exploration and Visualization (EDA)

We begin by exploring the dataset. After removing many features, it now consists of **8357 entries** and **49 attribute** columns, of which 45 are numerical and 4 are categorical.

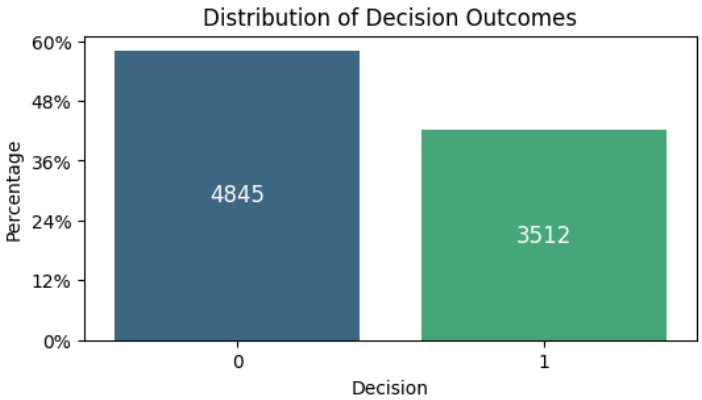
We also notice that the columns `samerace` and decision only contain values of 0 or 1, so they should be treated as binary features.
Additionally, we observe that the `field` attribute, one of the categorical features, has a very high number of unique values (high cardinality). Given the nature of this attribute, it might be a good idea to group the values into broader categories, as there is a clear and reasonable logic for doing so. This transformation will be applied during the Data Cleaning section.

Data Imbalance Analysis

One of the first questions we ask is whether the dataset is imbalanced.

By inspecting the target variable, we find that the value 0 appears 4,845 times (58%), while the value 1 appears 3,512 times (42%).

We conclude that, although the dataset is not perfectly balanced, it can still be considered **balanced** overall.

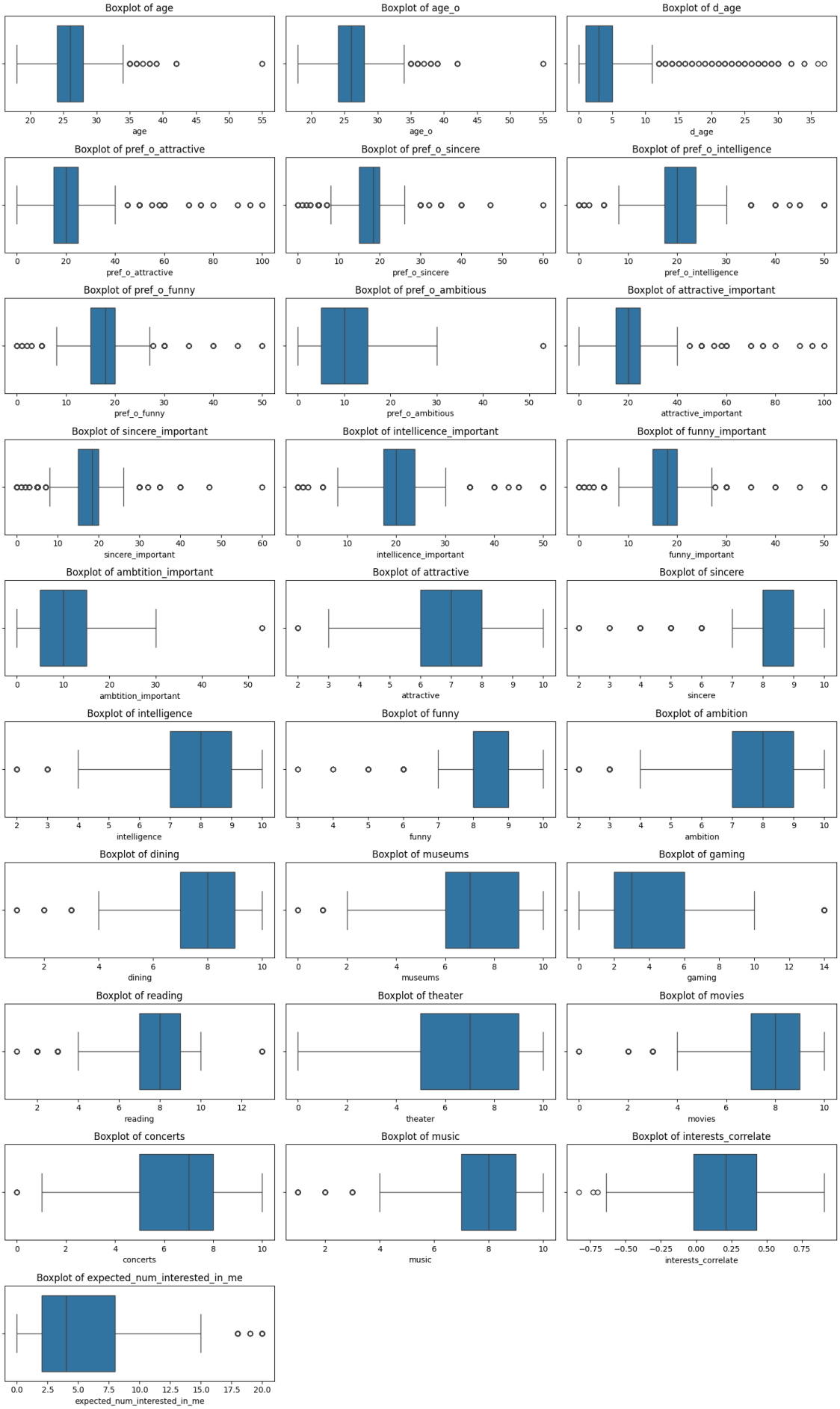


Check Outliers

To check whether the dataset contains outliers, we applied z-score normalization to all columns. If a value exceeds 3 (or -3) in absolute terms, it is considered an outlier.

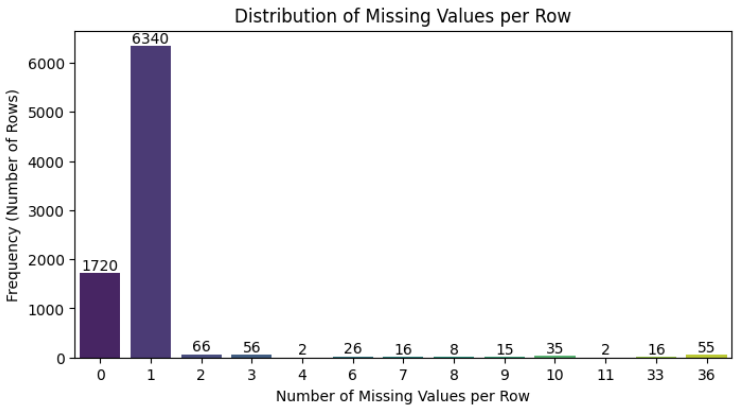
We found that **62%** of our numerical features **contain outliers**.

However, these unusual values should not be removed. They do not represent sensor errors or faulty data, instead, they are valid entries from potential users of our application who simply differ from the average. For example, the outliers in the age column, where we find users older than the average, still represent real users who would use the app. So, rather than removing them, we should ensure our models are trained on these data points as well.


Boxplot of age
age
Boxplot of age_o
age_o
Boxplot of d_age
d_age
Boxplot of pref_o_attractive
pref_o_attractive
Boxplot of pref_o_sincere
pref_o_sincere
Boxplot of pref_o_intelligence
pref_o_intelligence
Boxplot of pref_o_funny
pref_o_funny
Boxplot of pref_o_ambitious
pref_o_ambitious
Boxplot of attractive_important
attractive_important
Boxplot of sincere_important
sincere_important
Boxplot of intellicence_important
intellicence_important
Boxplot of funny_important
funny_important
Boxplot of ambtition_important
ambtition_important
Boxplot of attractive
attractive
Boxplot of sincere
sincere
Boxplot of intelligence
intelligence
Boxplot of funny
funny
Boxplot of ambition
ambition
Boxplot of dining
dining
Boxplot of museums
museums
Boxplot of gaming
gaming
Boxplot of reading
reading
Boxplot of theater
theater
Boxplot of movies
movies
Boxplot of concerts
concerts
Boxplot of music
music
Boxplot of interests_correlate
interests_correlate
Boxplot of expected_num_interested_in_me
expected_num_interested_in_me

Missing value statistics

Now we check the statistics related to NaN values in our dataset. We find a total of 9,995 NaN values (2.44%), and that 44 out of 49 features (89.80%) contain at least one missing value.



By inspecting the graph, we also see that most rows contain 0 or at most 1 NaN. Only a small number of rows have more than one missing value.

Data Visualization

Now we'll perform some visual analysis of the data to better understand the characteristics of the dataset and the relationships between variables.

Chart 1

This plot shows the distribution of date outcomes (`decision`) broken down by gender.

It reveals that female participants tend to give negative decisions more often than male participants, suggesting that women may be more selective when choosing a partner.

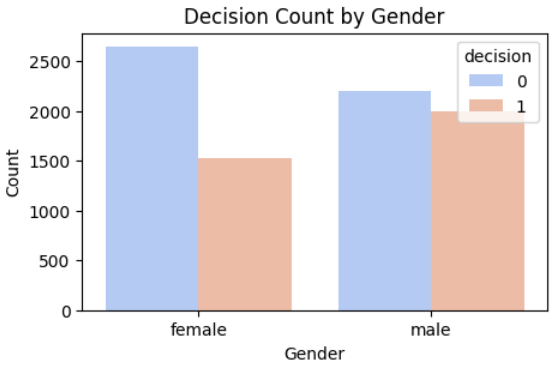
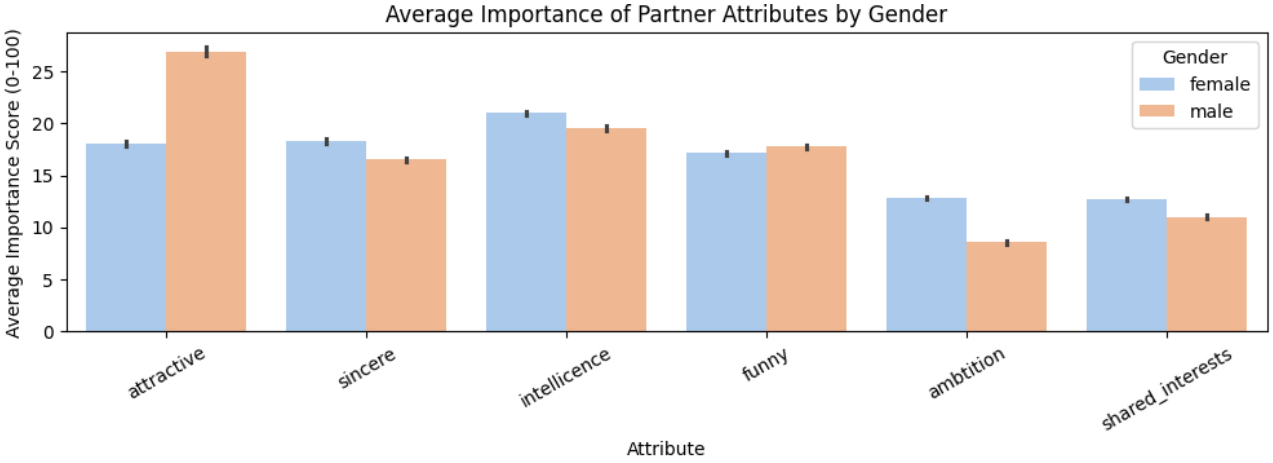


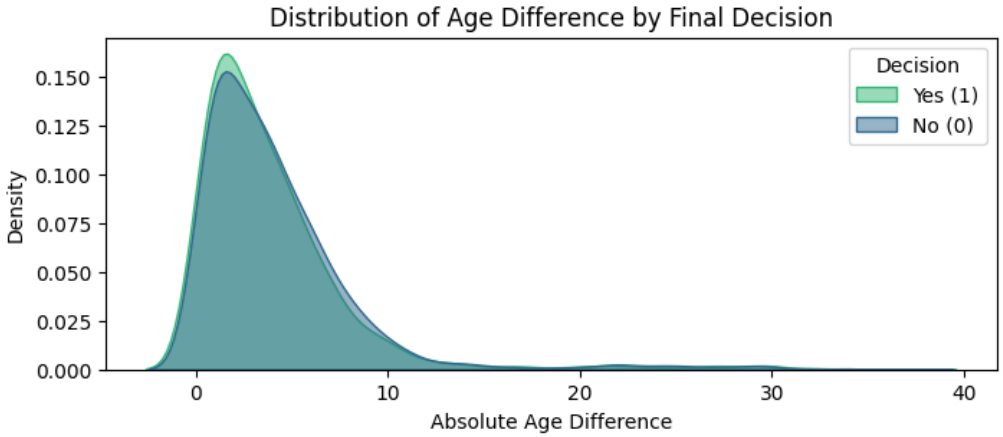
Chart 2



This plot indicates that men pay more attention to physical appearance when looking for a partner, whereas women tend to value sincerity and ambition more.

Chart 3

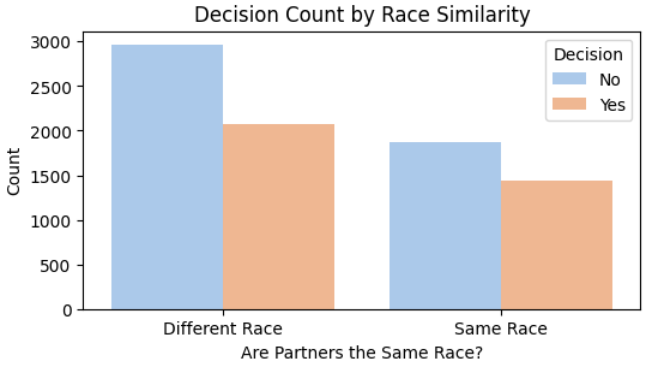
We now ask ourselves: does age difference matter when selecting a partner?



This chart shows us that age difference is not an important factor in this context.

Chart 4

Finally, we wondered whether being of the same race influences the decision.



In this case too, the answer is no: being of the same race does not have an impact.

Feature Engineering

In this phase, we finally got hands-on with the dataset. Specifically, we focused on fixing errors and transforming certain features to make them easier for the models to work with.

The first variable we modified was `field`. As mentioned earlier, it had too many unique values, so we decided that the best approach was to group them into broader professional categories. We grouped the 219 possible labels into 9 macro-categories: ArtsHumanities, Business, Education, Health, Law, Other, PublicAffairs, STEM, and SocialScience.

Data Correction

To avoid overloading this documentation, we will not go into the technical details of this section of the code. The main goal here was to check whether the features in the dataset contain only values that fall within the expected range, as defined in the official documentation provided with the dataset.

During this process, we encountered several anomalies, such as values outside their expected ranges. Some of these were due to typing errors, which we handled by replacing the incorrect values with the median of the respective columns. Other values, which couldn't be easily corrected or predicted, were simply replaced with NaN, trusting that the imputation algorithm would later handle them appropriately.

One example worth mentioning is the triplet of columns: age (participant's age), age_o (partner's age) and d_age (age difference). We identified a logical inconsistency: if either age or age_o was NaN, then d_age should also be NaN, yet this was often not the case. We fixed this inconsistency accordingly.

This data correction phase concluded with the removal of two columns:

- wave: since the wave-specific features were already normalized (using Min-Max scaling) and are no longer needed for modeling.
- expected_num_interested_in_me: because it contains 78.52% missing values, making it too sparse to be useful.

Data Cleaning

Before moving on to the imputation phase, we want to take one last look at the distribution of NaN values per row. To do this, we generate a table that shows how many NaNs each row contains and the percentage of rows with that exact number of missing values.

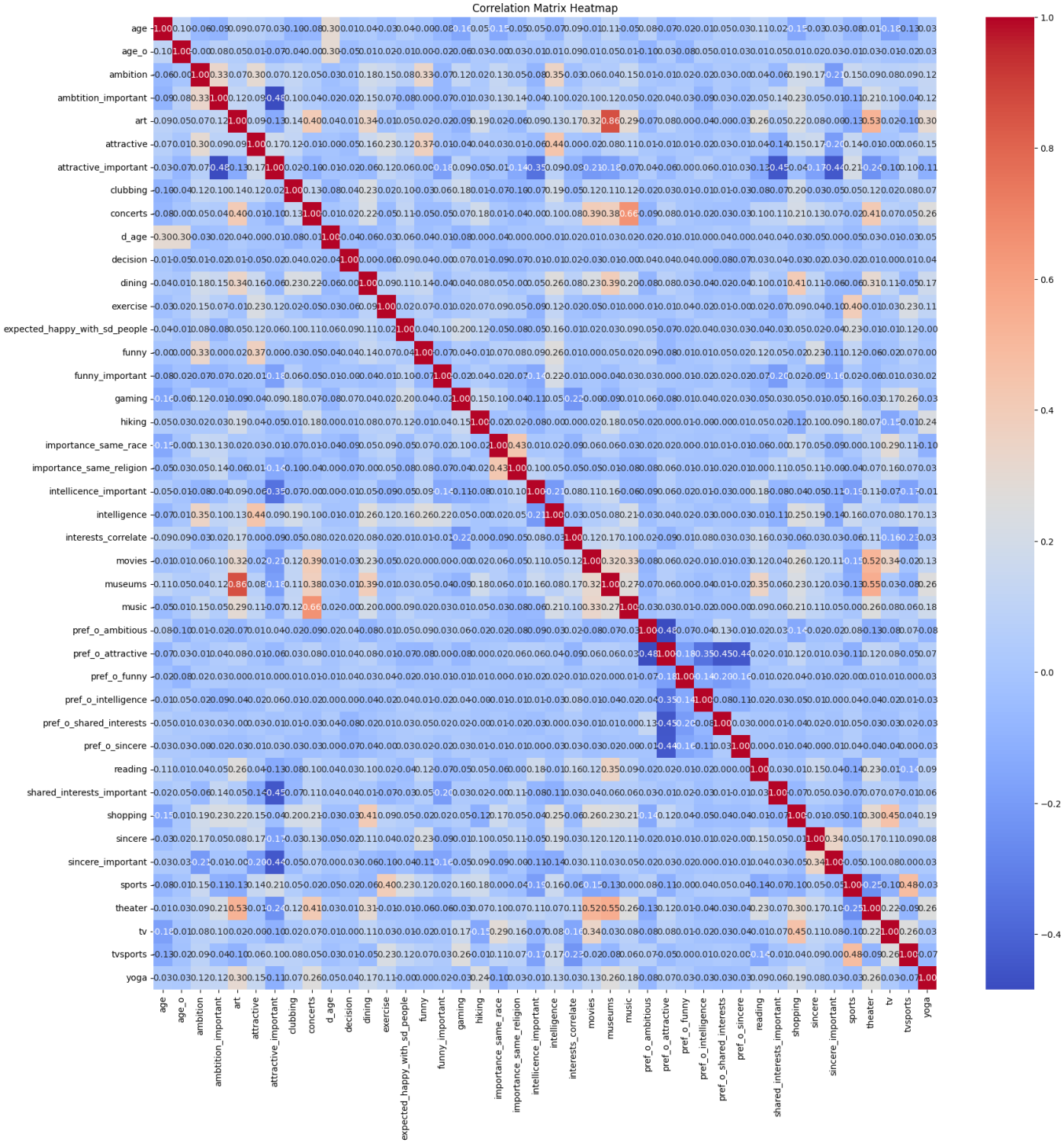
We notice that a small number of rows have too many NaNs. Since these are very few, we decide to remove them. Their presence, due to the high number of missing values, could negatively impact the quality of the classification model we aim to build.

Testing feature correlation

It's important to point out that, before running any tests to check for correlation between our features, we first applied an imputation algorithm, specifically, **Miss Forest**. We will discuss the imputation phase in detail in the next chapter.

Only after the dataset had been cleaned and the missing values imputed, we performed a correlation check by calculating the Pearson correlation matrix.

Looking at the features in the dataset, we expected to find some pairs with high correlation, for example, art and museums, or concerts and music. The correlation matrix confirms that these pairs are fairly correlated (they are the orange-colored squares we can see in the matrix) but not to a degree (such as 90%) that would justify removing one of the two features. So, we decided to keep the dataset as it is.


Correlation Matrix Heatmap
age
age_o
ambition
ambition_important
art
attractive
attractive_important
clubbing
concerts
d_age
decision
dining
exercise
expected_happy_with_sd_people
funny
funny_important
gaming
hiking
importance_same_race
importance_same_religion
intellicence_important
intelligence
interests_correlate
movies
museums
music
pref_o_ambitious
pref_o_attractive
pref_o_funny
pref_o_intelligence
pref_o_shared_interests
pref_o_sincere
reading
shared_interests_important
shopping
sincere
sincere_important
sports
theater
tv
tvsports
yoga
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4

Classification

Data Splitting

To evaluate the predictive performance of the model and prevent data leakage, the dataset was split into two subsets: **85%** for training and **15%** for testing. This **stratified** split ensures that the distribution of the target variable is preserved across both subsets.

Preprocessing phase

Before training a classification algorithm, it is essential to preprocess the raw dataset. This step is crucial because Machine Learning models require input data to be in a numeric, clean, and standardized format.

In its original form, our dataset presents several challenges: missing values, numerical features with very different scales, and categorical features in text format.

To address these challenges in a structured and reproducible way, we built a **preprocessing pipeline** using the scikit-learn libraries. The approach involves creating specific pipelines for each data type (`dtype`) in the dataset, which are then combined using a `ColumnTransformer`. This ensures that each column receives the most appropriate preprocessing.

- **Numerical Features**: For these columns, the pipeline performs two key steps:
  1. **Imputation**: Missing values were estimated using `IterativeImputer`, an advanced technique that preserves the relationships between variables.
  2. **Scaling**: The features were scaled with `RobustScaler` to standardize their ranges and make them robust to outliers.
- **Categorical Features**: Text-based columns like `field` were processed as follows:
  1. **Imputation**: Missing values were filled with the most frequent category.
  2. **Encoding**: The categories were converted into a numerical format using `OneHotEncoder`, which creates a new binary column for each unique value while preventing multicollinearity (`drop='first'`).
- **Binary Features**: Binary columns (e.g., `gender`, `samerace`) were normalized: textual labels ('female'/'male') were mapped to integer values (0/1), and any missing values were imputed with the most frequent value.

All these steps are orchestrated by the ColumnTransformer, which applies each specific logic to its corresponding set of columns. The final output is a homogeneous data matrix, entirely numerical and free of missing values, ready to be used as input for our classification algorithms.

Focus on the Imputation Algorithm – MissForest

Since we needed to perform imputation on both numerical and categorical columns, our initial choice was the **MissForest** algorithm (based on the Random Forest algorithm), which is a non-parametric and iterative method for missing value imputation. It is often considered a state-of-the-art solution in this area.

Unlike other methods that require separate treatment for different data types, MissForest handles them in an integrated and intelligent way. The key is that its "engine" is the Random Forest algorithm, which naturally deals with mixed feature types, acting as a classifier for categorical variables and as a regressor for numerical ones.
This algorithm is very effective, but its main drawback is execution time. We applied it to our dataset and then examined the imputed values. In our case, it did not produce values different from the mode for categorical features. Therefore, it doesn't make sense to keep using it, especially considering its high computational cost (it takes about 7 minutes per run).

Although MissForest likely performs better on numerical features than other algorithms, we realized it's more practical to sacrifice a bit of accuracy in exchange for faster processing. When we use pipelines for cross-validation, imputation will be repeated for every fold, and the total computation time will grow accordingly. So, we cannot afford to use such a slow algorithm.
At this point, we decided to use the mode for imputing categorical columns, since MissForest was basically doing the same. For numerical columns, we will use a much faster algorithm: the `IterativeImputer`.

Model Selection and Hyperparameter Tuning

To identify the most effective algorithm for predicting dating decisions, a systematic approach to model evaluation and optimization was employed. This process involves selecting a set of candidate models, defining a search space for their hyperparameters, and establishing clear criteria for measuring their performance.

Candidate Models

Four powerful, tree-based ensemble algorithms were selected for evaluation, as they are well-suited for structured, tabular data and are known for their high predictive performance.

The candidate estimators are:

- **HistGradientBoostingClassifier**: A modern, fast implementation of gradient boosting provided by Scikit-learn, optimized for large datasets.
- **XGBClassifier**: A highly optimized and popular gradient boosting library, renowned for its performance and flexibility.
- **CatBoostClassifier**: Another state-of-the-art gradient boosting framework, which includes native handling of categorical features and is known for its robustness.
- **RandomForestClassifier**: A classic ensemble method that builds multiple decision trees and aggregates their predictions to improve accuracy and control over-fitting.

Hyperparameter Search Space

A model's performance is highly dependent on its configuration. To find the optimal settings for each algorithm, a specific grid of hyperparameters (`param_grids`) was defined. This grid outlines the search space for the tuning process.

The search grids were designed to explore key parameters that influence model complexity, learning speed, and regularization, such as:

- **Learning Rate and Number of Estimators**: Controlling the trade-off between training speed and performance (learning_rate, max_iter, n_estimators, iterations).
- **Tree Structure**: Limiting the complexity of individual trees to prevent overfitting (max_depth, min_samples_leaf).
- **Handling Class Imbalance**: Using parameters like class_weight and scale_pos_weight to give more importance to the minority class, which is critical in datasets where one outcome is much rarer than the other.
- **Regularization**: Applying penalties to prevent the model from becoming too complex (l2_regularization, l2_leaf_reg, gamma).

Each combination of these hyperparameters will be tested for each model to identify the most effective configuration.

Evaluation and Selection Criteria

To ensure a robust and unbiased evaluation, model performance will be assessed using cross-validation. Multiple scoring metrics were defined to provide a comprehensive view of each model's strengths and weaknesses:

- **accuracy**: The proportion of correctly classified decisions. It provides a general measure of overall performance.
- **f1**: The harmonic mean of precision and recall. It is particularly useful for imbalanced datasets, as it provides a better measure of a model's performance on the minority class.
- **roc_auc**: The Area Under the ROC Curve. It measures the model's ability to distinguish between the positive and negative classes across all classification thresholds.

While all three metrics will be calculated for analysis, accuracy has been designated as the `refit_metric`. This means the hyperparameter combination that yields the highest average accuracy during cross-validation will be considered the "winner". The best model will then be re-trained on the entire training dataset using this optimal set of parameters.

Model Training and Evaluation

To systematically find the best-performing model and its optimal configuration, a comprehensive training and evaluation workflow was implemented. This process leverages `GridSearchCV` to automate hyperparameter tuning for each candidate model within a robust cross-validation framework.

The Automated Workflow

The entire process is encapsulated in a loop that iterates through each of the four candidate models. For each model, the following steps are executed:

1. **Pipeline Integration**: A complete Pipeline object is created, chaining the preprocessor (defined in the previous section) with the current model estimator. This ensures that data preprocessing is applied consistently and correctly within each cross-validation fold, preventing data leakage from the validation set into the training set.
2. **Cross-Validated Grid Search**: A `GridSearchCV` object is configured to perform an exhaustive search over the predefined hyperparameter grid for the model. The evaluation is conducted using a `StratifiedKFold` cross-validation strategy with 5 splits. Stratified folding is crucial here, as it preserves the percentage of samples for each class in the target variable (`decision`) across all folds, which is essential for reliable evaluation on potentially imbalanced datasets.
3. **Multi-Metric Evaluation**: The search evaluates each hyperparameter combination using the three specified metrics: `accuracy`, `f1`, and `roc_auc`. While all scores are recorded for analysis, accuracy is designated as the `refit metric`. This instructs `GridSearchCV` to identify the model with the highest average accuracy score and then automatically retrain it on the entire training dataset using the best-found parameters.
4. **Execution and Storage**: Upon completion, the following key artifacts are stored:
    - `best_estimators`: A dictionary containing the final, refitted pipeline for each model, ready for inference on new data.
    - `results`: A summary dictionary holding the best cross-validation score, the optimal parameters, and the total execution time for each model. This is used for a high-level comparison.
    - `all_cv_results`: A dictionary containing the complete cv_results_ DataFrame from GridSearchCV. This detailed log is invaluable for in-depth analysis, such as examining the

trade-offs between different hyperparameter settings and diagnosing overfitting by comparing training and validation scores.

5. **Immediate Feedback and Analysis**: After each model's run, a detailed table is displayed, showing the top-performing hyperparameter combinations. This table includes the mean scores for all metrics on both the internal cross-validation test folds (`mean_test_*`) and the training folds (`mean_train_*`). Comparing these scores provides immediate insight into model behavior and helps identify if a particular configuration is overfitting the training data (i.e., showing a large gap between training and test performance).

Performance Table from GridSearchCV for the Different Methods

For the sake of brevity, we report here only the performance tables on the test set for the two best-performing models. The tables are sorted by test set accuracy (from cross-validation), which is the metric we use to determine the best model.

1. GridSearchCV Results for **Hist Gradient Boosting**:

mean test accuracy	mean test f1	mean test roc-auc	mean train accuracy	mean train f1	mean train roc-auc	Learning rate	Max depth	Class weight	Max iter	l2 regularization
0.704812	0.654249	0.763989	0.886163	0.869587	0.958043	0.05	9	balanced	180	1.0
0.702540	0.651874	0.764174	0.895889	0.880645	0.964269	0.05	9	balanced	200	1.0
0.701972	0.650285	0.764576	0.905509	0.891507	0.970006	0.05	9	balanced	220	1.0

2. GridSearchCV Results for **Catboost**

mean test accuracy	mean test f1	mean test roc-auc	mean train accuracy	mean train f1	mean train roc-auc	Iterations	l2_leaf_reg
0.701404	0.652386	0.763021	0.863056	0.842790	0.941688	340	6
0.700693	0.652222	0.765405	0.864227	0.844466	0.943165	340	5
0.699558	0.650895	0.761809	0.856560	0.835287	0.936486	320	6

Results

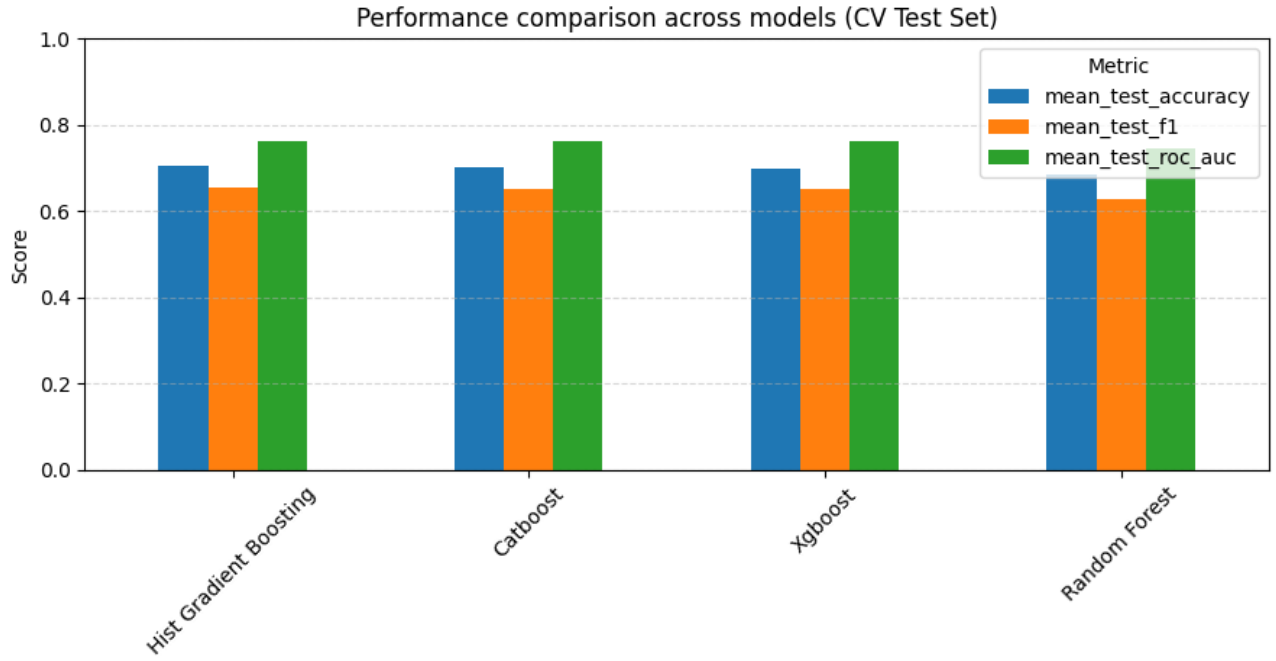
After running the exhaustive `GridSearchCV` for each of the four candidate models, the next crucial step is to consolidate and compare their performance in a clear and concise way.

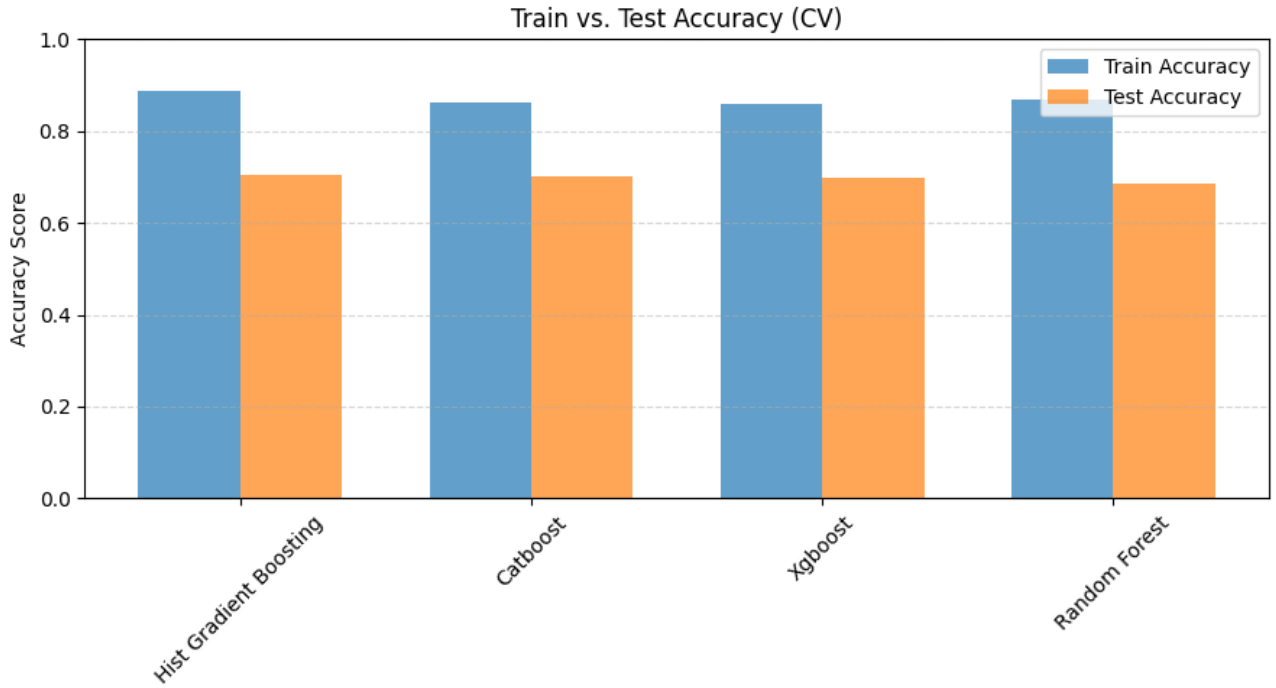
The code loops through the saved results of each model. For each one, it selects the single best hyperparameter combination by finding the entry with the highest rank (`rank_test_accuracy`), which represents the best-performing configuration according to our chosen `refit metric`.

We then obtain a final summary table, sorted by `mean_test_accuracy`, to identify the overall best model based on cross-validation performance.

Model	Mean test accuracy	Mean test f1	Mean test roc-auc	Mean train accuracy	Mean train f1	Mean train roc-auc
Hist Gradient Boosting	0.7048	0.6542	0.7640	0.8862	0.8696	0.9580
Catboost	0.7014	0.6524	0.7630	0.8631	0.8428	0.9417
Xgboost	0.6994	0.6516	0.7634	0.8601	0.8394	0.9390

Random Forest	0.6866	0.6287	0.7443	0.8691	0.8477	0.9478





The table shows that the **HistGradientBoostingClassifier** is the best-performing model, outperforming all other candidates across the three main evaluation metrics on the validation set (the internal test sets from cross-validation).

Interestingly, **CatBoost** and **XGBoost** follow very closely behind the winner, with almost identical performance. Both reach an accuracy of around 70%, with only a slight drop compared to HistGradientBoosting. The **RandomForestClassifier**, on the other hand, ranks slightly lower, with an accuracy of 68.7%.

Observations on Overfitting

A key insight that emerges from the table is the degree of overfitting observed in all models. Performance on the training set (`mean_train_accuracy` between 86% and 89%) is significantly higher than on the validation set (`mean_test_accuracy` around 70%).

This suggests that the models, with their optimized hyperparameters, are quite complex and have "memorized" a large portion of the training data.

Model Validation: A Rigorous Statistical Comparison

While the initial cross-validation results provided a ranking of the models, it is important to recognize that these performance metrics are point estimates and are subject to variability based on the specific data splits. The **HistGradientBoostingClassifier** emerged as the top performer, but **CatBoost** followed very closely. To make a definitive and statistically sound decision between these two contenders, a more rigorous head-to-head comparison was conducted on the **unseen hold-out test set**.

Statistical Testing Methodology

A paired statistical testing procedure was designed to rigorously compare the models. This approach is powerful because it evaluates both models on the exact same subsets of data, reducing variance and isolating the performance difference attributable to the models themselves.

Step 1: Resampling with RepeatedStratifiedKFold

Instead of a single train/test split, we used RepeatedStratifiedKFold. This technique involves:

1. **Splitting**: The test set is divided into 10 stratified folds.
2. **Evaluating**: The models are trained on 9 folds and evaluated on the 10th. This is repeated 10 times until each fold has been used for evaluation once.
3. **Repeating**: The entire 10-fold cross-validation process is repeated 50 times, each time with a new random shuffling of the data.

This process yields 500 pairs of performance scores (one for `HistGradientBoosting`, one for `CatBoost`) for each metric. This large sample of results provides a highly stable estimate of each model's true generalization performance.

Step 2: Calculating Paired Differences

For each of the 500 evaluation runs, we calculated the difference in performance between the two models (e.g., `Accuracy_CatBoost` - `Accuracy_HistGradientBoosting`). This resulted in a list of 500 "difference" values. Our statistical test will analyze this distribution of differences.

Step 3: The Assumption Check - Testing for Normality

Many powerful statistical tests, including the paired t-test, assume that the data they are analyzing follows a normal (or "Gaussian") distribution. Before proceeding with the t-test, we must validate this assumption.

- **Method**: The **Shapiro-Wilk test** was used to check if the distribution of the 500 performance differences was normal.
- **Hypothesis**: The null hypothesis (H_0) of the Shapiro-Wilk test is that the data is drawn from a normal distribution.
- **Interpretation**: A p-value **greater than 0.05** suggests that we do not have enough evidence to reject the null hypothesis. In this case, we can safely assume normality and proceed with the t-test.

Step 4: The Hypothesis Test - Paired T-Test

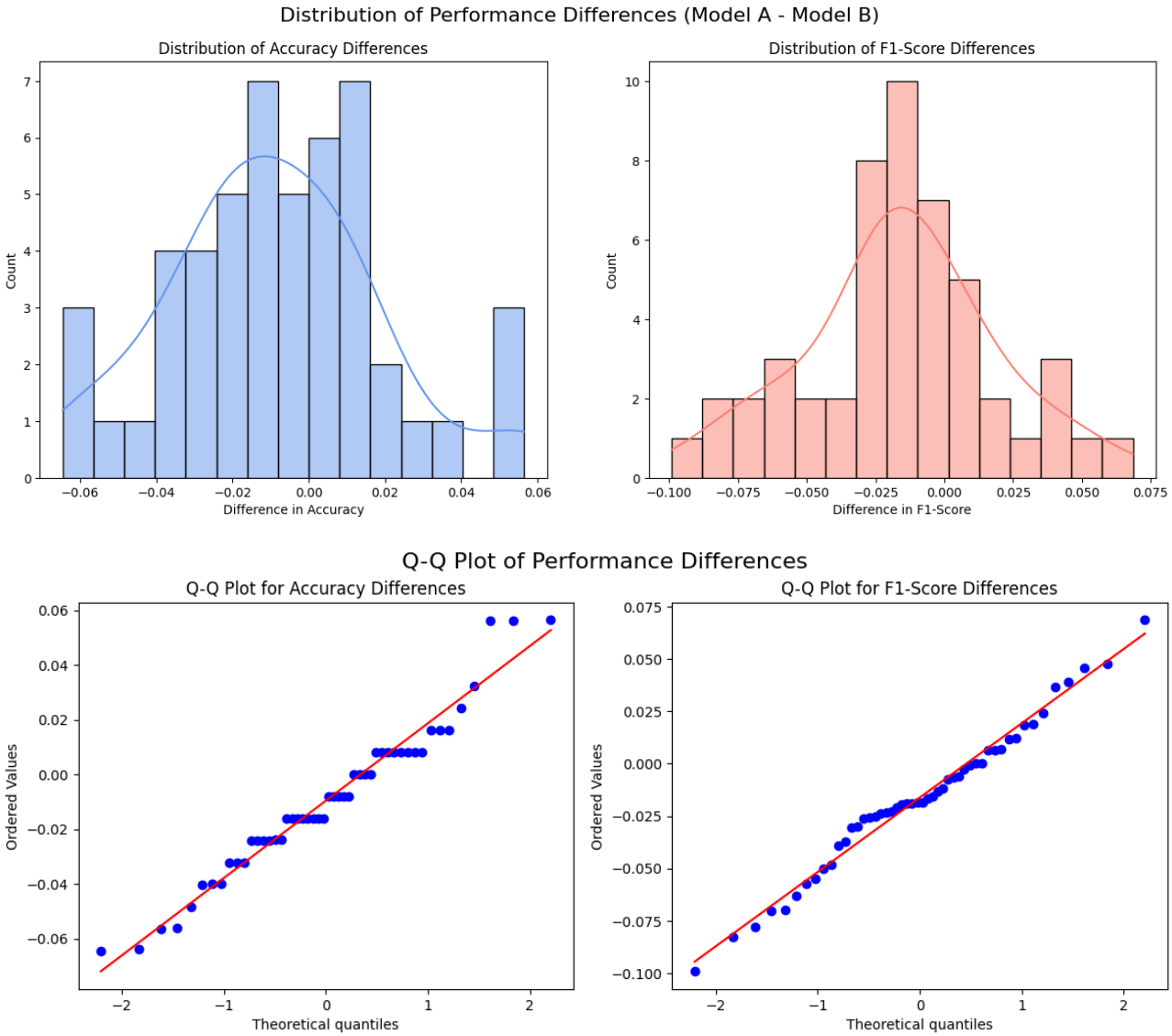
Once normality was confirmed, the **paired t-test** was performed on the distribution of differences.

- **Hypothesis**: The null hypothesis (H_0) for the paired t-test is that the true mean difference in performance between the two models is zero (i.e., they perform equally well on average). The alternative hypothesis (H_1) is that the mean difference is not zero.
- **Interpretation**: A p-value **less than 0.05** provides strong evidence to reject the null hypothesis. It indicates that the observed average difference in performance is statistically significant and unlikely to be due to random sampling variability.

Results and Interpretation

The results of this rigorous procedure were clear and decisive:

Normality Test Results:



- The Shapiro-Wilk test on the accuracy differences yielded a **p-value of 0.1446**
- The Shapiro-Wilk test on the F1-score differences yielded a **p-value of 0.7959**

Since both p-values are well above 0.05, the normality assumption holds for both metrics, validating the use of the paired t-test.

Paired T-Test Results:

- **Accuracy**: The t-test resulted in a p-value of 0.0185
- **F1-Score**: The t-test resulted in a p-value of 0.0019

Both p-values are significantly below the 0.05 threshold. This allows us to confidently **reject the null hypothesis** in both cases. The performance advantage demonstrated by `CatBoost` over `HistGradientBoosting` on the test set is not a random artifact but a statistically significant finding.

Final Conclusion: Selecting CatBoost

Despite `HistGradientBoosting` showing a slight edge during the initial cross-validation phase, the more robust statistical analysis on the hold-out test set unequivocally demonstrates that `CatBoost` is the superior model for this task.

It not only offers a significant advantage in training efficiency (as seen in the previous section) but also delivers a **statistically significant improvement in both accuracy and F1-score** when generalizing to completely unseen data. Therefore, `CatBoost` **is selected as the final, champion model for this project**.

Final Evaluation of the Champion Model on the Test Set

Having identified `CatBoost` as the superior model through rigorous cross-validation and statistical testing, the final step is to evaluate its performance on the hold-out test set. This dataset was kept separate throughout the entire training and tuning process and serves as the ultimate benchmark for the model's ability to generalize to new, real-world data.

The model used for this evaluation is the `CatBoost` pipeline that was already retrained on the full training dataset using its optimal hyperparameters, which are listed below for full transparency:

Optimal Hyperparameters:

- 'depth': 6,
- 'iterations': 340,
- 'l2_leaf_reg': 6,
- 'learning_rate': 0.05,
- 'random_strength': 1.0,
- 'scale_pos_weight': np.float64(1.3745785569790965),
- 'subsample': 0.8

Performance Metrics

The model's predictions on the test set were compared against the true outcomes, yielding the following key performance indicators:

Metric	Score	Interpretation
Accuracy	0.7039	The model correctly predicts the user's decision (Match/No Match) in **70.4%** of cases.
F1-Score	0.6630	A strong F1-score, indicating a healthy balance between precision and recall.
ROC-AUC Score	0.7859	Excellent discriminative power; the model is very good at distinguishing a positive from a negative outcome.

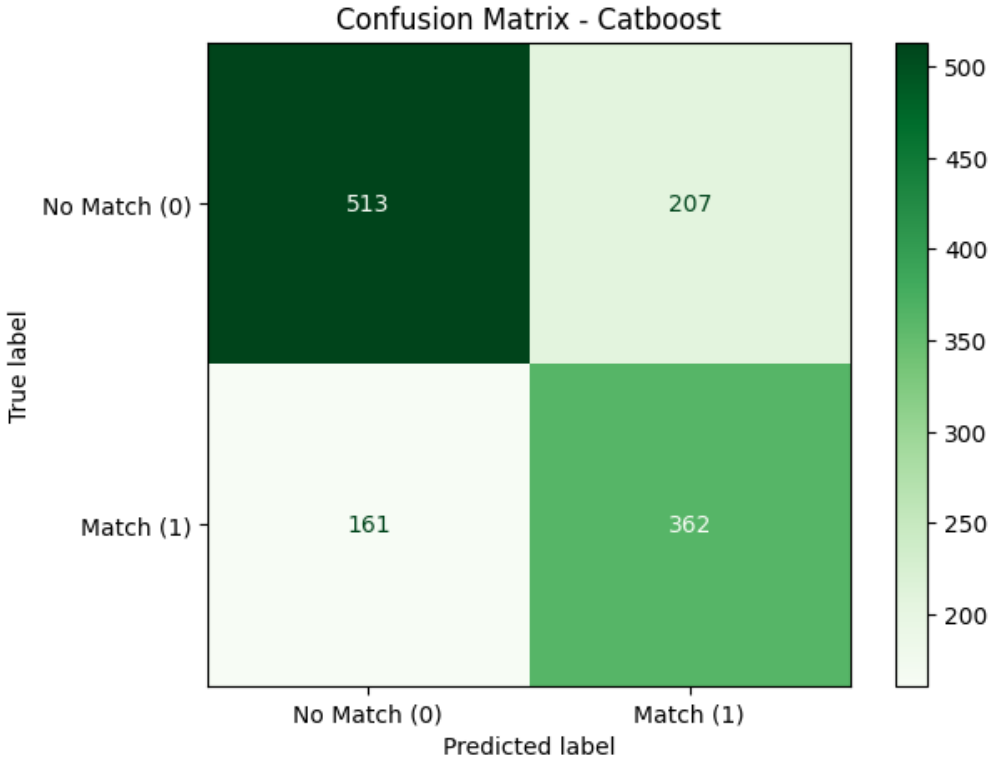
Classification Report Analysis

```
Classification Report on Test Set:
              precision    recall  f1-score   support

No Match (0)       0.76      0.71      0.74       720
   Match (1)       0.64      0.69      0.66       523

    accuracy                           0.70      1243
   macro avg       0.70      0.70      0.70      1243
weighted avg       0.71      0.70      0.71      1243
```

- **"No Match" (Class 0)**: The model is highly reliable when predicting a "No Match," with a **precision of 0.76**. This means that when the model decides to filter out a profile, it is correct 76% of the time.
- **"Match" (Class 1)**: Performance on the positive class is also solid. With a **precision of 0.64**, the model is correct in nearly two-thirds of its positive recommendations. The **recall of 0.69** means it successfully identifies 69% of all potential matches, a strong result for a recommendation system.



The final evaluation confirms that the `CatBoost` **model is a robust and effective classifier for this task**. With an overall accuracy of **70.4%**, it provides a solid foundation for an intelligent dating app recommendation engine. It successfully identifies a majority of compatible partners while effectively filtering out a large number of incompatible ones, thereby significantly improving the user experience compared to a random or non-personalized approach.

Further Analysis

In light of the results obtained, we want to try adding some additional preprocessing techniques to our pipeline, with the goal of improving the current performance.

PCA

To investigate whether dimensionality reduction could improve model performance, reduce overfitting, or decrease computational cost, we integrated Principal Component Analysis (PCA) in our preprocessing pipeline.

Our methodology involved extending the `GridSearchCV` process to optimize PCA's `n_components` hyperparameter alongside the models' own hyperparameters. The grid for `n_components` was set to `[0.90, 0.95, 40, 50, None]`. This allowed us to test configurations that:

- Retain 90% or 95% of the total variance.
- Reduce the feature space to a fixed number of 40 or 50 components.
- Serve as a control by using all original components (None), effectively disabling PCA's reduction aspect.

The experiment was conducted on our primary ensemble models: HistGradientBoostingClassifier and CatBoostClassifier.

Results & Analysis

The results from the cross-validation were unequivocal and demonstrated a negative impact from using PCA:

- **Performance Degradation**: Both models experienced a significant drop in predictive accuracy when PCA was applied. For instance, the best cross-validation accuracy for HistGradientBoosting dropped from approximately **0.705** (without PCA) to **0.666** (in the PCA-inclusive grid). A similar decrease was observed for CatBoost.
- **Optimal** `n_components`: Across all top-performing runs, GridSearchCV consistently identified `n_components=None` as the optimal parameter. This empirically confirms that maintaining the full, original feature set is superior to any of the tested reduction strategies.
- **Increased Overfitting**: An interesting side effect was an increase in the gap between training and validation accuracy. This suggests that the models, while attempting to compensate for the transformed feature space, settled on more complex configurations that overfit the training data.

Our initial expectation was that PCA would help our models by simplifying the data. By reducing the number of features, we hoped to see better generalization on the test set, meaning higher accuracy and less overfitting.
However, the results showed the opposite. While the models' performance on the **training data did, in fact, improve**, their performance on the **test data got significantly worse**. This widened gap between training and test scores is a classic sign of **increased overfitting**.

So, why did PCA make our models overfit more?
The issue stems from how PCA interacts with our tree-based models. PCA creates new, blended features that obscure the specific, non-linear patterns that algorithms like Gradient Boosting excel at finding. By removing what it deems as low-importance (low-variance) information, PCA may have discarded subtle but critical predictive signals. This created a simplified version of the training data that the model could easily "memorize," leading to high training scores. However, because this simplified view did not capture the true complexity of the data, the model failed to generalize to the unseen test set.

In essence, PCA provided the models with a simplified, distorted view of the data. The models became experts at understanding this distorted view but were then unable to cope with the complexity of the real-world data, leading to poor performance and higher overfitting.

Final Decision

Given the clear empirical evidence that PCA harms model performance for this specific problem and model set, we have made the decision to **exclude PCA from the final production pipeline**. Our modeling efforts will continue using the complete, preprocessed feature set, which has proven to yield the best generalization performance.

Feature Selection

Since our experiment with Principal Component Analysis did not deliver the expected performance gains, we will now pivot to a more straightforward dimensionality reduction strategy: feature selection. This approach does not transform the data but instead identifies and retains a subset of the most impactful original features. To accomplish this, we will utilize the `SelectKBest` method, testing it with a range of different k values (the number of features to select) to determine the optimal feature subset for our models.

We integrated SelectKBest into our GridSearchCV pipeline, testing the following hyperparameters:

- **Number of Features** (k): We tested selecting the top 40 features, top 50 features, and using 'all' features as a baseline control.
- **Scoring Function** (`score_func`): We evaluated two different statistical tests:
    - `f_classif` (ANOVA F-test), which captures linear relationships between features and the target.
    - `mutual_info_classif`, which can capture any kind of statistical dependency, including non-linear relationships.

Results & Analysis

The cross-validation results provided a clear and consistent outcome across both the HistGradientBoosting and CatBoost models.

- **HistGradientBoosting**: The top-performing configurations all resulted from the baseline setting where k = 'all'. This means that using the complete feature set yielded the highest accuracy (~0.705), outperforming any subset created by SelectKBest. The best-performing model with a reduced feature set (k=40) achieved a slightly lower accuracy of ~0.701.
- **CatBoost**: The results for CatBoost were more nuanced but led to a similar conclusion. The best overall performance was achieved with k=50 and the f_classif scoring function, which yielded an accuracy of ~0.702. This represents a marginal improvement over the baseline model's accuracy of ~0.701. However, other top-ranking models also used 'all' features, indicating that the benefit of feature selection was minimal and not consistently superior.

Final Decision

As largely anticipated, the configurations that retained all features (k='all') were consistently among the top performers for both models. While feature selection with SelectKBest did not degrade performance as significantly as PCA, it also failed to provide a compelling improvement.

For the CatBoost model, the slight accuracy gain (from ~0.701 to ~0.702) by selecting 50 features is marginal. When considering the trade-off, we introduce the additional complexity and potential instability of a feature selection step for a negligible performance benefit. Maintaining the full feature set ensures

that the models have access to all potentially useful information, which is a key strength of robust algorithms like Gradient Boosting.

Therefore, to prioritize maximum performance and model stability, we have decided to **exclude feature selection from our final pipeline**. The evidence from both our PCA and `SelectKBest` experiments strongly suggests that for this particular dataset, our tree-based models perform optimally when trained on the complete set of original features.

Feature Discretization as an Overfitting Reduction Strategy

Our comprehensive experiments with dimensionality reduction, using both PCA (feature extraction) and SelectKBest (feature selection), have conclusively shown that these techniques do not improve our models' performance. In both cases, the optimal configurations involved retaining the full, original feature set, suggesting that our tree-based classifiers are adept at handling the initial feature space without modifications.

However, a key challenge remains: our best-performing models still exhibit a degree of overfitting, learning the training data more effectively than they generalize to unseen data. This leaves us with one final avenue to explore for performance enhancement: **feature discretization**.

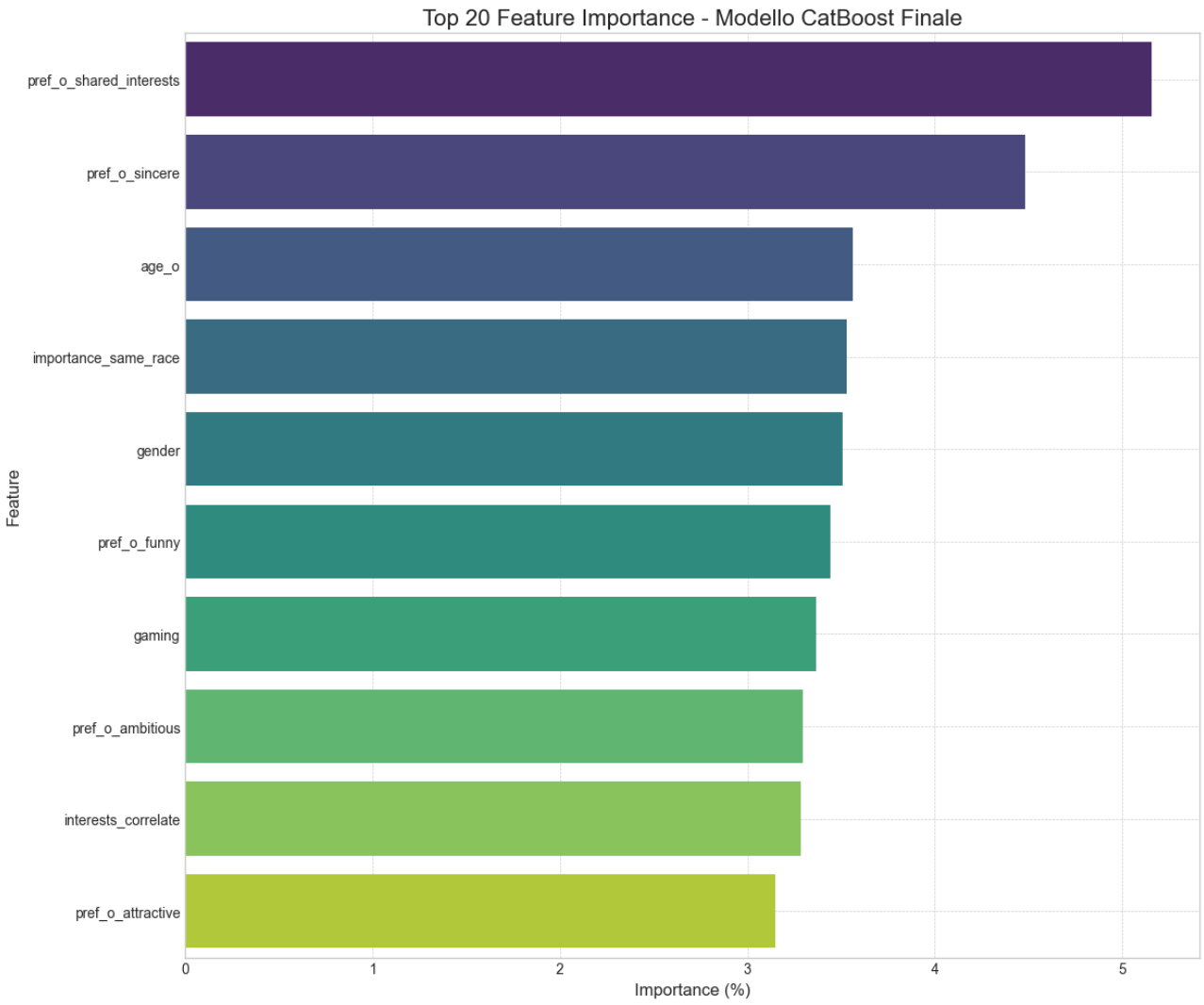
The core idea is to transform our most important continuous numerical features into discrete categorical bins. By doing this, we aim to prevent the models from "over-specializing" on the precise numerical values within the training set. Instead of learning complex granular rules, the model will learn more robust, generalized rules.

Feature important

We then took our best model, CatBoost with the best possible hyperparameter configuration, and used it to generate a chart showing the top 20 most important features.

Next, we selected the top 3 most important features and added a preprocessing step to the pipeline, using scikit-learn's KBinsDiscretizer to discretize the numerical variables into bins (intervals), and then converting them into one-hot encoded categorical features.

As shown in the chart, the 3 features in question are: [`'pref_o_shared_interests'`, `'pref_o_sincere'`, `'age_o'`].



Results

Unfortunately, this targeted modification also failed to produce the desired improvements. The performance did not drop as significantly as with PCA, but the results were slightly worse than our baseline, offering no benefit in terms of accuracy or overfitting reduction.

Here is a summary of the performance with discretization applied:

Model	**Mean Test Accuracy**	**Mean Test F1**	**Mean Test ROC-AUC**	**Mean Train Accuracy**
Hist Gradient Boosting	0.6986	0.6473	0.7604	0.8900
CatBoost	0.6959	0.6452	0.7604	0.8547

When comparing these results to our best models without discretization (which achieved test accuracies of ~0.705 for HistGB and ~0.701 for CatBoost), it is clear that discretization led to a marginal decrease in performance.

Conclusion

At this point, having explored multiple avenues for dimensionality reduction and feature engineering, we have concluded that for this dataset, our tree-based models perform optimally on the original, complete

feature set. Since the discretization step provides no real benefit and only increases the pipeline's complexity and computational cost, we have decided to **remove this final modification**. Our definitive approach will be to proceed with the best-performing models trained on all a-priori features.