Open to (Psychedelic) Experience

By Jordan Loewen-Colón May 30th 2023

The Business Problem

The (fictional) MindSpectrum Research Institute, is deeply engaged in groundbreaking work involving the clinical trials of psychedelic-assisted therapies. These trials seek to gauge the safety and effectiveness of various psychedelic substances, including psilocybin, MDMA, LSD, Ketamine, and Cannabis. These substances, when used in conjunction with psychotherapy, are being tested as potential treatments for various mental health disorders, such as depression, anxiety, post-traumatic stress disorder (PTSD), and addiction.

Our task is to assist the institute with its patient recruitment campaign, aiming to attract individuals who are not only afflicted by the targeted conditions, but are also open to the concept of psychedelic usage. The goal is to identify individuals who are naturally inclined towards trying psychedelics, without the need for excessive persuasion or influence.

In this relatively nascent field of psychedelic science, it's critical for the institute to maintain a positive public image, hence there's an inherent need to ensure that trial participants are optimally suited. The ideal participant would display an inherent willingness to try psychedelics, indicating a certain level of curiosity and a potentially positive mindset, which in turn may contribute to improved trial outcomes.

Our data science problem is to develop a predictive model focusing on 'precision' as the key performance indicator, aiming to minimize false positives in identifying potential trial participants. The institute prefers a focused approach, favoring a smaller, more reliable group of

Our Recommendations?

The Mind Spectrum Institute should prioritize incorporating Oscore assessment into screening processes which could improve predictions of psychedelic use, as higher scores often indicate an inclination towards such usage. Given the noticeable difference in average Oscores between psychedelic users (0.152) and non-users (-0.593), investigating Oscore's influence on therapeutic effects of psychedelic-assisted therapies could yield valuable insights. And, finally, the study should taget those who've never used legal highs, nicotine, or amyl nitrites as potential participants for psychedelic trials.

Step 1: Data Understanding

To make our recommendations, we analyzed the Drug Consumptions (UCI) (https://www.kaggle.com/datasets/obeykhadija/drug-consumptions-uci) from Kaggle. As stated on the original database:

"The Database contains records for 1885 respondents. For each respondent 12 attributes are known: Personality measurements which include NEO-FFI-R (neuroticism, extraversion, openness to experience, agreeableness, and conscientiousness), BIS-11 (impulsivity), and ImpSS (sensation seeking), level of education, age, gender, country of residence and ethnicity. All input attributes are originally categorical and are quantified. After quantification values of all input features can be considered as real-valued. In addition, participants were questioned concerning their use of 18 legal and illegal drugs (alcohol, amphetamines, amyl nitrite, benzodiazepine, cannabis, chocolate, cocaine, caffeine, crack, ecstasy, heroin, ketamine, legal highs, LSD, methadone, mushrooms, nicotine and volatile substance abuse and one fictitious drug (Semeron) which was introduced to identify over-claimers. For each drug they have to select one of the answers: never used the drug, used it over a decade ago, or in the last decade, year, month, week, or day."

We begin by importing the proper tools and then the data itself.

In [1]:
```
#Import libraries
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import math

from sklearn.model_selection import train_test_split, GridSearchCV, cross_val_
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifie
from sklearn.metrics import classification_report, confusion_matrix, recall_sc
from scipy.stats import uniform, randint, pointbiserialr, ttest_ind
from tabulate import tabulate

%matplotlib inline

# Import the data
df = pd.read_csv(r'Data/Drug_Consumption.csv', header= 0,
                         encoding= 'unicode_escape')
```

In [2]:
```
# Print the first five rows
df.head()
```

Out[2]:

	ID	Age	Gender	Education	Country	Ethnicity	Nscore	Escore	Oscore	AScore	...	E
0	2	25-34	M	Doctorate degree	UK	White	-0.67825	1.93886	1.43533	0.76096	...	
1	3	35-44	M	Professional certificate/ diploma	UK	White	-0.46725	0.80523	-0.84732	-1.62090	...	
2	4	18-24	F	Masters degree	UK	White	-0.14882	-0.80615	-0.01928	0.59042	...	
3	5	35-44	F	Doctorate degree	UK	White	0.73545	-1.63340	-0.45174	-0.30172	...	
4	6	65+	F	Left school at 18 years	Canada	White	-0.67825	-0.30033	-1.55521	2.03972	...	

5 rows × 32 columns

In [3]:
```
# Check Data
print(df.shape)
df.info()
```

```
(1884, 32)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1884 entries, 0 to 1883
Data columns (total 32 columns):
 #   Column     Non-Null Count  Dtype  
---  ------     --------------  -----  
 0   ID         1884 non-null   int64  
 1   Age        1884 non-null   object 
 2   Gender     1884 non-null   object 
 3   Education  1884 non-null   object 
 4   Country    1884 non-null   object 
 5   Ethnicity  1884 non-null   object 
 6   Nscore     1884 non-null   float64
 7   Escore     1884 non-null   float64
 8   Oscore     1884 non-null   float64
 9   AScore     1884 non-null   float64
 10  Cscore     1884 non-null   float64
 11  Impulsive  1884 non-null   float64
 12  SS         1884 non-null   float64
 13  Alcohol    1884 non-null   object 
 14  Amphet     1884 non-null   object 
 15  Amyl       1884 non-null   object 
 16  Benzos     1884 non-null   object 
 17  Caff       1884 non-null   object 
 18  Cannabis   1884 non-null   object 
 19  Choc       1884 non-null   object 
 20  Coke       1884 non-null   object 
 21  Crack      1884 non-null   object 
 22  Ecstasy    1884 non-null   object 
 23  Heroin     1884 non-null   object 
 24  Ketamine   1884 non-null   object 
 25  Legalh     1884 non-null   object 
 26  LSD        1884 non-null   object 
 27  Meth       1884 non-null   object 
 28  Mushrooms  1884 non-null   object 
 29  Nicotine   1884 non-null   object 
 30  Semer      1884 non-null   object 
 31  VSA        1884 non-null   object 
dtypes: float64(7), int64(1), object(24)
memory usage: 471.1+ KB
```

The dataset has 1884 entries and 31 columns with a mix of floats and objects. The personality scores (6:12) is measured on a Likert-based scale ranging from 0 (“Strongly Disagree”) to 4 (“Strongly Agree”) and then rendered as a float. The demographics have various sub categories, and the drug values are measured by recency (if ever) the substance has been consumed; CL0 being never used, and CL6 being used in the last day.

In [4]:
```
# Check for missing values
print(f"\n {'Nulls in Column'.title()} \n {df.isnull().sum()}")

# Check for duplicate values
print(f"\n {'Duplicates'.title()} :- {len(df.loc[df.duplicated()])}")
```

```
 Nulls In Column
 ID           0
Age           0
Gender        0
Education     0
Country       0
Ethnicity     0
Nscore        0
Escore        0
Oscore        0
AScore        0
Cscore        0
Impulsive     0
SS            0
Alcohol       0
Amphet        0
Amyl          0
Benzos        0
Caff          0
Cannabis      0
Choc          0
Coke          0
Crack         0
Ecstasy       0
Heroin        0
Ketamine      0
Legalh        0
LSD           0
Meth          0
Mushrooms     0
Nicotine      0
Semer         0
VSA           0
dtype: int64

 Duplicates :- 0
```

The data looks pretty clean! No duplicates or null values!

Looking at the data descriptions, a possible point of interest is the drug category "Semeron." The data collectors created this fictitous class of drug to weed out people who would over identify drug use as a control. Checking the values of that column, it looks like there were only about 8 over-claimers.

In [5]:
```
## Semeron values
print(df['Semer'].value_counts())
```

```
CL0    1876
CL2       3
CL3       2
CL1       2
CL4       1
Name: Semer, dtype: int64
```

It also might be worth taking a look at the means and standard deviations of our personality trait columns.

In [6]:
```
# Select columns 5 to 12
selected_columns = df.iloc[:, 5:13]

# Calculate mean and standard deviation
mean_values = selected_columns.mean()

# Display the results
print("Mean values:")
print(mean_values)
```

```
Mean values:
Nscore      -0.000119
Escore       0.000143
Oscore      -0.000225
AScore       0.000242
Cscore      -0.000383
Impulsive    0.007335
SS          -0.002667
dtype: float64
```

That Sensation Seeking scores highest in mean value seems to make sense.

Step 2: Data Preperation

Since we are looking to target just psychedelic drugs, we will create a column that only includes those drugs considered under the broad definition of psychedelics: cannabis, ecstasy, ketamine, LSD, and mushrooms.

In [7]:
```
#Create our Target Column
df_p = df.copy()
df_p = df_p.drop(columns=['ID'])
df_p['Psychedelics'] = ''
Psychedelics = ['Cannabis','Ecstasy','Ketamine','LSD','Mushrooms']

# Create a function that determines whether or not someone has consumed a psyc
for i in range(0, len(df_p)):
    tot = 0
    for n in Psychedelics:
        if df_p[n][i] != "CL0":
            tot = tot + 1
        if tot > 0:
            df_p['Psychedelics'].iat[i] = 1
        else:
            df_p['Psychedelics'].iat[i] = 0
```

In [8]:
```
#Drop unnecessary columns
df_p = df_p.drop(columns=['Cannabis','Ecstasy','Ketamine','LSD','Mushrooms'],
df_p.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1884 entries, 0 to 1883
Data columns (total 27 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   Age           1884 non-null   object
 1   Gender        1884 non-null   object
 2   Education     1884 non-null   object
 3   Country       1884 non-null   object
 4   Ethnicity     1884 non-null   object
 5   Nscore        1884 non-null   float64
 6   Escore        1884 non-null   float64
 7   Oscore        1884 non-null   float64
 8   AScore        1884 non-null   float64
 9   Cscore        1884 non-null   float64
 10  Impulsive     1884 non-null   float64
 11  SS            1884 non-null   float64
 12  Alcohol       1884 non-null   object
 13  Amphet        1884 non-null   object
 14  Amyl          1884 non-null   object
 15  Benzos        1884 non-null   object
 16  Caff          1884 non-null   object
 17  Choc          1884 non-null   object
 18  Coke          1884 non-null   object
 19  Crack         1884 non-null   object
 20  Heroin        1884 non-null   object
 21  Legalh        1884 non-null   object
 22  Meth          1884 non-null   object
 23  Nicotine      1884 non-null   object
 24  Semer         1884 non-null   object
 25  VSA           1884 non-null   object
 26  Psychedelics  1884 non-null   object
dtypes: float64(7), object(20)
memory usage: 397.5+ KB
```

In [9]:
```
#Check values
print(df_p['Psychedelics'].value_counts())
```

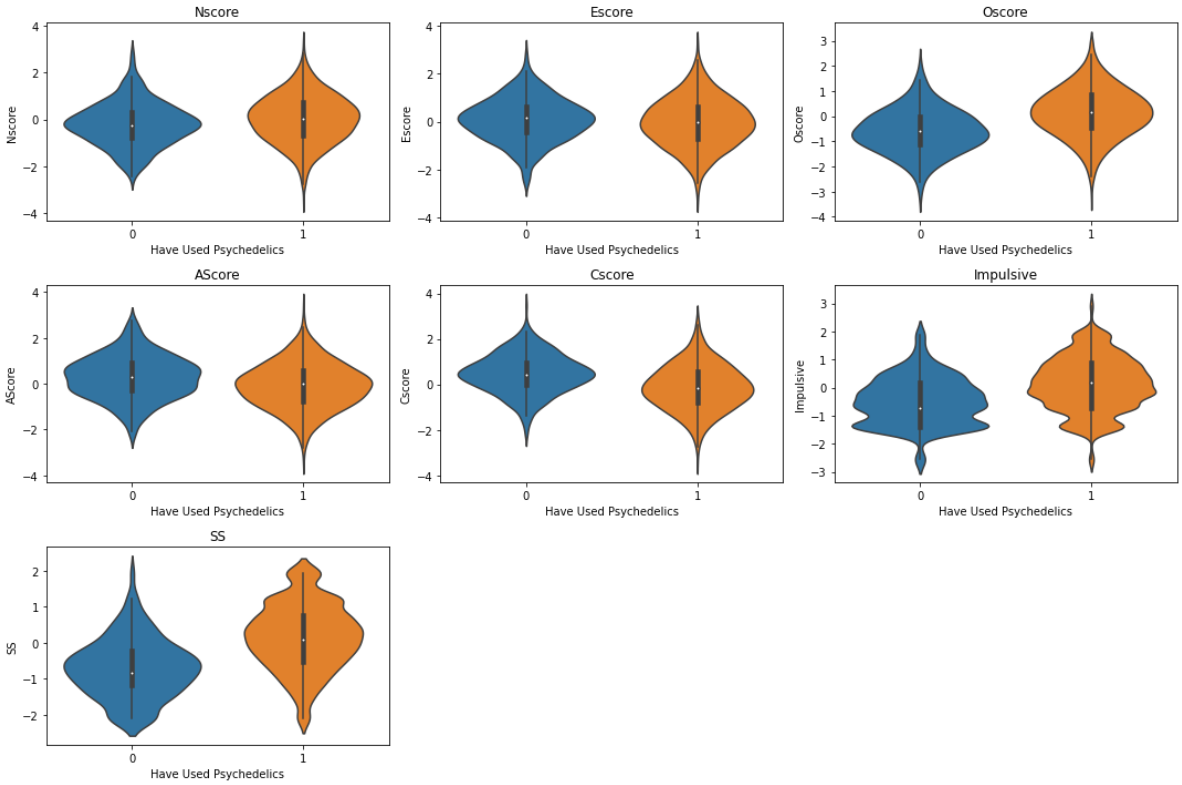
```
1    1494
0     390
Name: Psychedelics, dtype: int64
```

In [10]:
```
#drop any rows of individuals claiming to take Semer
df_p.drop(df_p.loc[df_p['Semer']!='CL0'].index, inplace=True)
print(df_p['Semer'].value_counts())
```

```
CL0    1876
Name: Semer, dtype: int64
```

Since we are primarily focused on understanding psychedelic use based on personality scores, let's make a quick set of violin plots to get a sense of the connection.

In [11]:
```
personality_scores = df_p.columns[5:12]

# Define the number of columns and rows for the subplots matrix
num_cols = 3  # Number of columns
num_rows = math.ceil(len(personality_scores) / num_cols)  # Number of rows

# Create the subplots matrix
fig, axes = plt.subplots(num_rows, num_cols, figsize=(15, 10))

# Flatten the axes array to simplify indexing
axes = axes.flatten()

# Iterate over each numerical column
for i, column in enumerate(personality_scores):
    ax = axes[i]
    sns.violinplot(x=df_p['Psychedelics'], y=df_p[column], ax=ax)
    ax.set_xlabel('Have Used Psychedelics')
    ax.set_ylabel(column)
    ax.set_title(f'{column}')

# Hide any unused subplots
for j in range(len(personality_scores), len(axes)):
    axes[j].axis('off')

# Adjust spacing between subplots
plt.tight_layout()

# Show the plots
plt.show()
```



Based on these plots, we might expect there to be some strong correlation between high scorers on "Open-to-Experience" (Oscore), "Sensation Seeking" (SS), and Impulsiveness.

Our next steps are to split the data into our training and test sets, and then create a pipeline to streamline and organize our code, enhancing readability and reproducibility.

In [12]:
```
#create our train test split
y = df_p['Psychedelics']
y = y.astype('int')
X = df_p.drop(columns=['Psychedelics'], axis=1)

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
```

In [13]:
```
# Print out the sizes to verify
shape_table = [['Original', X.shape, y.shape], ['Training', X_train.shape, y_t
         ['Testing', X_test.shape, y_test.shape]]
print(tabulate(shape_table, headers=['Dataset', 'X shape', 'y shape']))
```

```
Dataset    X shape     y shape
---------  ----------  ---------
Original   (1876, 26)  (1876,)
Training   (1407, 26)  (1407,)
Testing    (469, 26)   (469,)
```

Create Pipeline

In [14]:
```
# Create the preprocessing steps for numerical data
subpipe_num = Pipeline(steps=[('num_impute', SimpleImputer()),
                              ('ss', StandardScaler())])

# Create the preprocessing steps for categorical data
subpipe_cat = Pipeline(steps=[('cat_impute', SimpleImputer(strategy='most_freq
                               ('ohe', OneHotEncoder(sparse=False, handle_unknow

# Combine the preprocessing steps for numerical and categorical data
CT = ColumnTransformer(transformers=[('subpipe_num', subpipe_num, [5, 6, 7, 8,
                                                ('subpipe_cat', subpipe_cat, [0, 1, 2
                                                                             18, 19, 20,
                              remainder='passthrough')
```

Step 3 - Modeling

Logistic Regression

Our first model will be a simple logistic regression. Starting with a logistic regression model offers interpretability and simplicity, serving as an efficient method to establish baseline performance for binary classification, such as distinguishing participants willing to try

psychedelics. Its probabilistic output and capability to highlight feature importance can provide crucial insights into factors influencing willingness to participate in the trial, while setting a comparative standard for future, more complex models.

In [15]:
```
# Create the logistic regression model
logistic_regression_model = LogisticRegression(max_iter=10000)

# Update the pipeline to include the logistic regression model
log_pipeline = Pipeline([
    ('preprocess', CT),
    ('classifier', logistic_regression_model)
])

# Fit the pipeline to the training data
log_pipeline.fit(X_train, y_train)

# Get the predicted labels for the training data
y_train_pred = log_pipeline.predict(X_train)

# Compute the precision of the logistic regression model
precision = precision_score(y_train, y_train_pred)

print(f"Precision: {precision}")
```

```
Precision: 0.9532374100719424
```

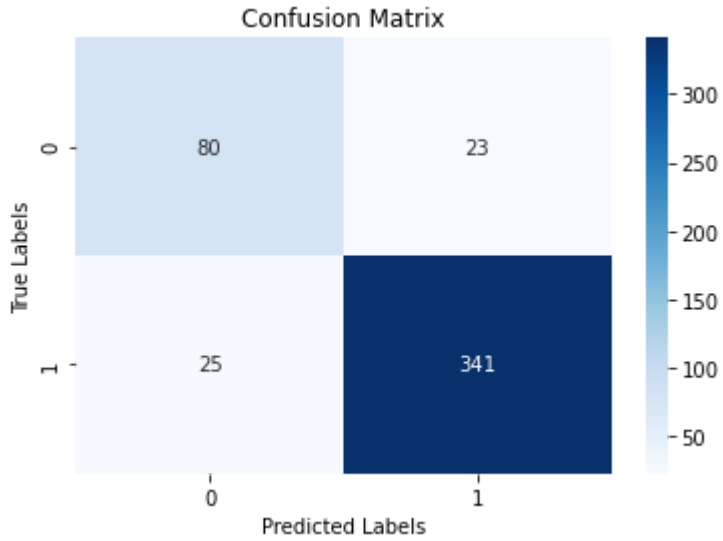
The log model has 95% precision on our training data, which implies a lower rate of false positives, as precision is the ratio of true positives to the sum of true positives and false positives. Now let's check the model on our test set:

In [16]:
```
# Predict the labels for the test data
y_test_pred = log_pipeline.predict(X_test)

# Print the classification report
print(classification_report(y_test, y_test_pred))
```

```
              precision    recall  f1-score   support

           0       0.76      0.78      0.77       103
           1       0.94      0.93      0.93       366

    accuracy                           0.90       469
   macro avg       0.85      0.85      0.85       469
weighted avg       0.90      0.90      0.90       469
```

Our model did great on the test set as well! Achieving a 94% precission rate on determining whether a person has taken any psychedelic. It's precision wasn't as good at predicting "no" to psychedelic use, but that's less important here. Next step is to create a confusion matrix to see how many false positives and negatives we had.

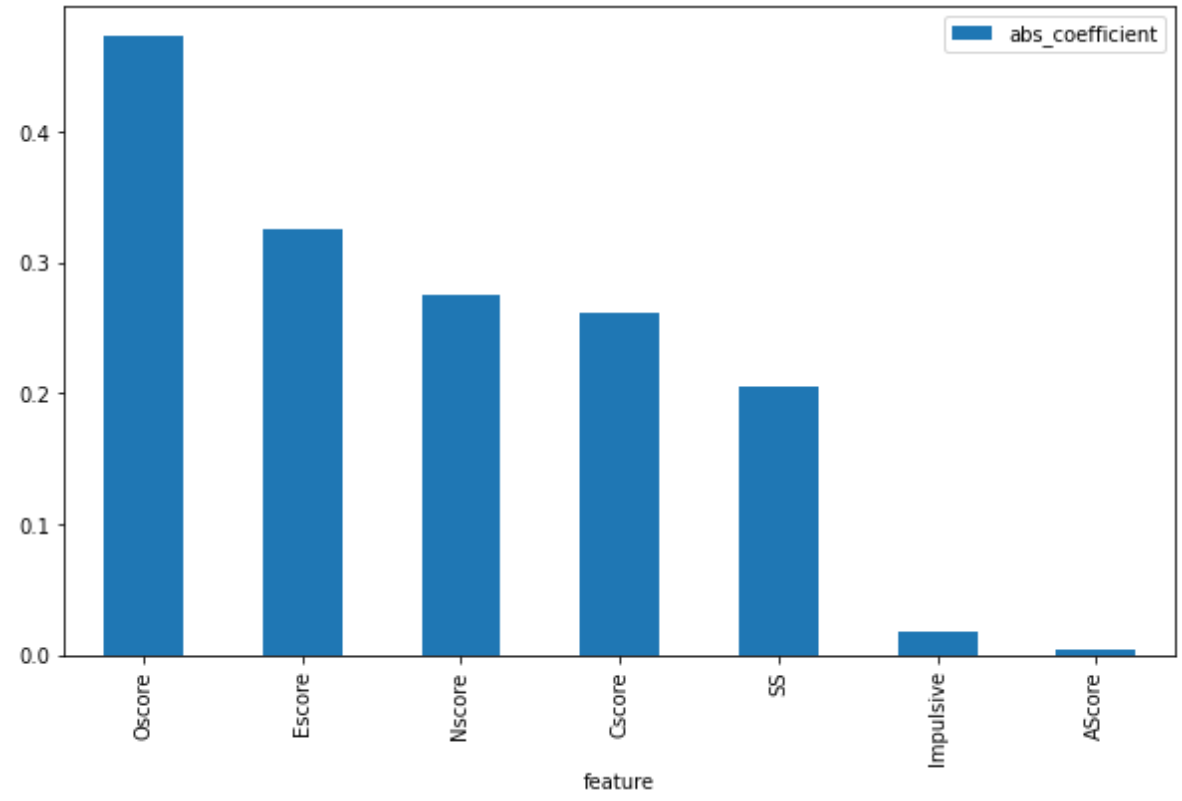
In [17]:
```
# Predict the labels for the test set
y_pred = log_pipeline.predict(X_test)

# Create the confusion matrix
cm = confusion_matrix(y_test, y_pred)

# Create a heatmap of the confusion matrix
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')

# Set labels, title, and axis ticks
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix')
plt.show()
```



Looks like the model had a total of 23 false positives. We will keep that in mind as we explore the other models.

Now it's time to explore the data. Our team needs to grab the feature importances in order to see which attributes the model thinks are more impotant than the others.

In [47]:
```
# Get column names after OneHotEncoding
cat_cols_transformed = log_pipeline.named_steps['preprocess'].named_transforme

# Concatenate with numerical column names to get the final order
feature_names_ordered = np.concatenate([X_train.columns[5:12], cat_cols_transf

# Get feature coefficients from the logistic regression model
coefficients = log_pipeline.named_steps['classifier'].coef_[0]

# Create a DataFrame with feature names and their coefficients
coefficients_df = pd.DataFrame({'feature': feature_names_ordered, 'coefficient

# Sort DataFrame by coefficient magnitude
coefficients_df['abs_coefficient'] = abs(coefficients_df['coefficient'])
coefficients_df = coefficients_df.sort_values(by='abs_coefficient', ascending=

# Filter DataFrame to include only desired features
num_features_df_log = coefficients_df[coefficients_df['feature'].isin(X_train.

# Plot the feature importances for the numerical features
num_features_df_log.plot(kind='bar', x='feature', y='abs_coefficient', figsize
```



It looks like Oscore has the largest coefficient magnitude of all our personality traits. The coefficient value of 0.5 for "Oscore" means that for every one-unit increase, the log odds of the outcome "Psychedelics" being 'yes' (versus 'no') increase by 0.5, assuming all other variables in the model are held constant.

To better understand this in terms of odds (rather than log odds), we can calculate the odds by taking the exponent of the coefficient: exp(0.5) ≈ 1.65. This means that for every one-unit increase in "Oscore", the odds of the outcome "Psychedelics" being 'yes' (versus 'no') increase by about 65%, assuming all other variables in the model are held constant.

And since, as we saw above, people who have taken psychedelics have a higher Oscore, we

Random Forest Classifier

Next we run a Random Forest Classifier, or RFC. It's worth running this model due to its ability to manage overfitting, handle missing values, deal with non-linear relationships, provide feature importance, deliver high accuracy, and its versatile application to both classification and regression tasks.

In [45]:
```
# Create the random forest classifier model
random_forest_model = RandomForestClassifier()

# Update the pipeline to include the random forest classifier model
rfc_pipeline = Pipeline([
    ('preprocess', CT),  # Preprocessing steps remain the same
    ('classifier', random_forest_model)  # Replace with random forest classifi
])

# Fit the pipeline to the training data
rfc_pipeline.fit(X_train, y_train)

# Predict the labels for the training data
y_train_pred_rfc = rfc_pipeline.predict(X_train)

# Compute the precision of the random forest model
precision_rfc = precision_score(y_train, y_train_pred_rfc)

print(f"RFC Precision: {precision_rfc}")
```

```
RFC Precision: 1.0
```

A precision of 1.0 indicates that our model is probably overfitting. To avoid that, we can run a RandomizedSearchCV to perform some hypertuning and get the best parameters for our model. Since we are targeting precision, we might want to adjust class weight values, but let's run the model first.

In [46]:
```
# Define the parameter grid for the random forest
param_dist = {
    'classifier__n_estimators': [100, 200, 500, 1000],
    'classifier__max_depth': [5, 10, 20],
    'classifier__min_samples_split': [10, 20, 30],
    'classifier__min_samples_leaf': [4, 8, 12],
    'classifier__max_features': ['sqrt', 'log2'],
    'classifier__bootstrap': [True]
}

# Create the RandomizedSearchCV object
random_search = RandomizedSearchCV(rfc_pipeline, param_distributions=param_dis

# Fit the RandomizedSearchCV object to the data
random_search.fit(X_train, y_train)

# Get the best parameters
best_params = random_search.best_params_

print("Best parameters:", best_params)
```

```
Best parameters: {'classifier__n_estimators': 500, 'classifier__min_samples_s
plit': 10, 'classifier__min_samples_leaf': 4, 'classifier__max_features': 'sq
rt', 'classifier__max_depth': 20, 'classifier__bootstrap': True}
```

In [48]:
```
# Create a new model with the best parameters
best_rfc = RandomForestClassifier(n_estimators=best_params['classifier__n_esti
                                  max_depth=best_params['classifier__max_depth
                                  min_samples_split=best_params['classifier__m
                                  min_samples_leaf=best_params['classifier__mi
                                  max_features=best_params['classifier__max_fe
                                  bootstrap=best_params['classifier__bootstrap

# Update the pipeline to include the new random forest classifier model
best_rfc_pipeline = Pipeline([
    ('preprocess', CT),
    ('classifier', best_rfc)
])

# Fit the pipeline to the training data
best_rfc_pipeline.fit(X_train, y_train)

# Predict the labels for the training data using the best model
y_train_pred_best = random_search.predict(X_train)

# Compute the precision of the best model
precision_best = precision_score(y_train, y_train_pred_best)

print(f"Best Model Training Precision: {precision_best}")
```

```
Best Model Training Precision: 0.9577836411609498
```

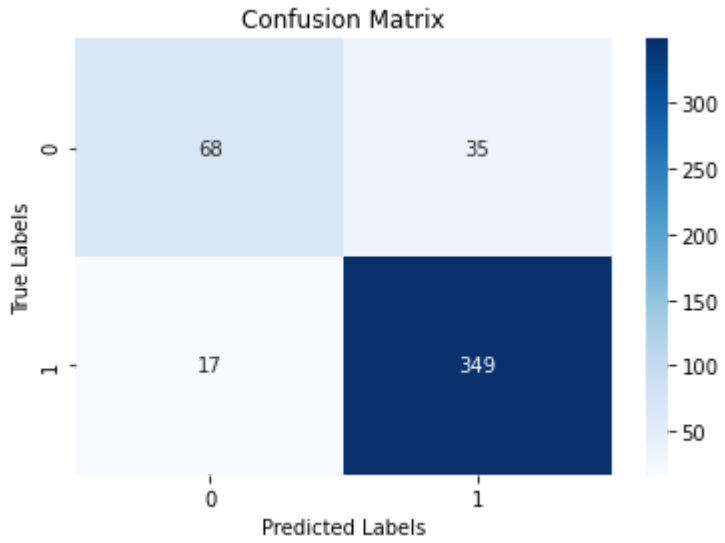
A model precision of 95 is great! Now let's run it on the test set:

In [49]:
```
# Predict the labels for the test data
y_test_pred = best_rfc_pipeline.predict(X_test)

# Print the classification report for test data
print(classification_report(y_test, y_test_pred))
```

```
              precision    recall  f1-score   support

           0       0.81      0.66      0.73       103
           1       0.91      0.96      0.93       366

    accuracy                           0.89       469
   macro avg       0.86      0.81      0.83       469
weighted avg       0.89      0.89      0.89       469
```

So it looks like our RFC scored only a 91 on our test set. Let's check the confusion matrix:

In [50]:
```
# Predict the labels for the test set using the best model
y_pred = random_search.predict(X_test)

# Create the confusion matrix
cm = confusion_matrix(y_test, y_pred)

# Create a heatmap of the confusion matrix
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')

# Set labels, title, and axis ticks
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix')
plt.show()
```

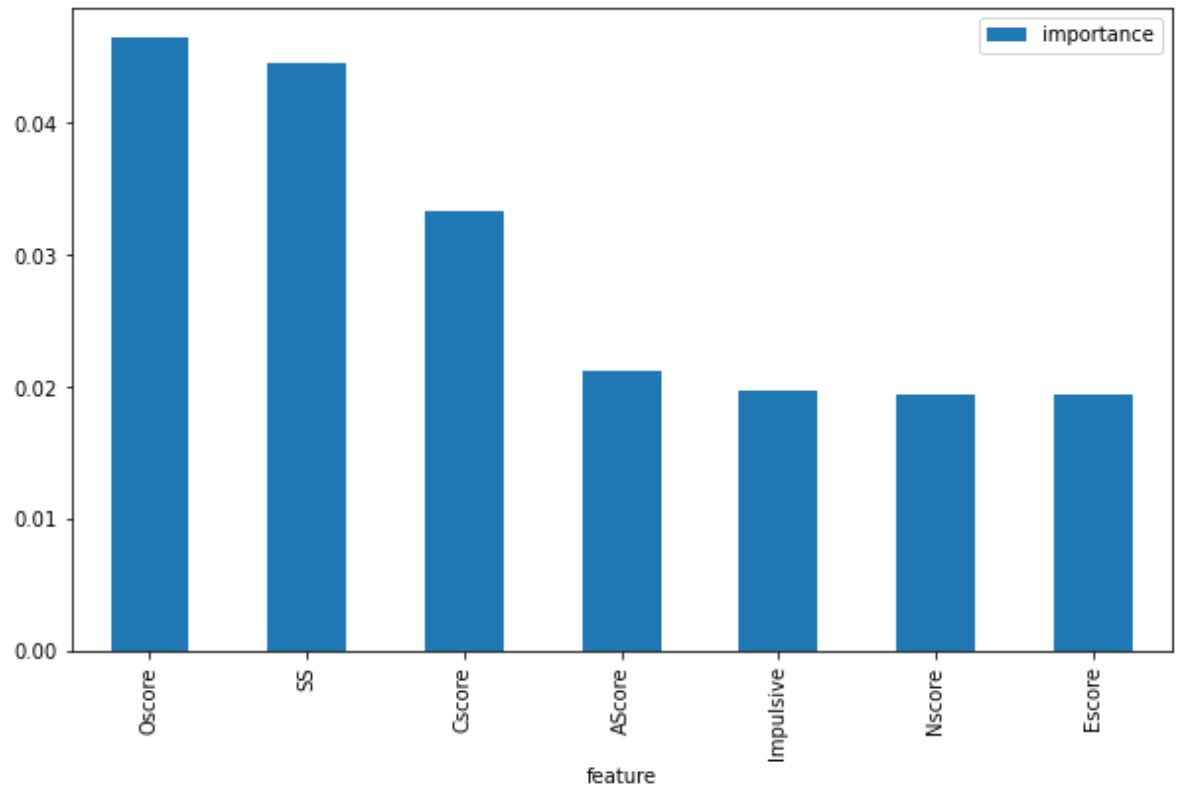


Rather than 23 false positives that our Log Model made, it looks like our RFC has 34. That's potentially 11 people who would have been falsely selected as likely to be willing to take psychedelics. Not great!

Let's see what this model has to say in terms of feature importances.

Get Feature Importances

In [51]:
```
# Get column names after OneHotEncoding
cat_cols_transformed = random_search.best_estimator_.named_steps['preprocess']

# Concatenate with numerical column names to get the final order
feature_names_ordered = np.concatenate([X_train.columns[5:12], cat_cols_transf

# Get feature importances from the best RFC
importances = random_search.best_estimator_.named_steps['classifier'].feature_

# Create a DataFrame with feature names and their importance
importances_df = pd.DataFrame({'feature': feature_names_ordered, 'importance':

# Sort DataFrame by importance
importances_df = importances_df.sort_values(by='importance', ascending=False)

# Filter DataFrame to include only desired features
num_features_df_rfc = importances_df[importances_df['feature'].isin(X_train.cc

# Plot the feature importances for the numerical features
num_features_df_rfc.plot(kind='bar', x='feature', y='importance', figsize=(10,
```



Just like our log model, the RFC seems to think Oscore is the most important, though the range between traits is much smaller here. It also tagged Impulsiveness pretty low.

But before we make any recommendations, we will check one more model.

Gradient Boosting Classifier

Its worth running a GBC model because they are known for handling overfitting well and can provide important insights into feature importance, contributing to model interpretability, complementing insights from RFC and Logistic Regression models.

In [25]:
```
# Define the Gradient Boosting Classifier pipeline
gbc_pipeline = Pipeline([
    ('preprocess', CT),
    ('classifier', GradientBoostingClassifier(random_state=42))
])

# Fit the pipeline on the training data
gbc_pipeline.fit(X_train, y_train)

# Predict the labels for the training data
y_train_pred = gbc_pipeline.predict(X_train)

# Compute the precision score
precision = precision_score(y_train, y_train_pred)

print(f"GBC Training Precision: {precision}")
```

```
GBC Training Precision: 0.9686940966010733
```

The initial model gives us a 97 precision! Great! But let's see if we can make it any better with some tuning.

In [30]:
```
# Define the parameter distribution for the Gradient Boosting Classifier
param_dist = {
    'classifier__n_estimators': randint(50, 200),
    'classifier__learning_rate': [0.01, 0.1, 1],
    'classifier__max_depth': randint(1, 40),
    'classifier__min_samples_split': randint(2, 11),
    'classifier__min_samples_leaf': randint(1, 5),
    'classifier__subsample': [0.5, 0.75, 1]
}

# Create the RandomizedSearchCV object
random_search_gbc = RandomizedSearchCV(gbc_pipeline, param_dist, n_iter=100, c

# Fit the RandomizedSearchCV object to the data
random_search_gbc.fit(X_train, y_train)

# Get the best parameters
best_params_gbc = random_search_gbc.best_params_

print("Best parameters:", best_params_gbc)
```

```
Best parameters: {'classifier__learning_rate': 0.1, 'classifier__max_depth':
3, 'classifier__min_samples_leaf': 1, 'classifier__min_samples_split': 6, 'cl
assifier__n_estimators': 191, 'classifier__subsample': 1}
```

In [35]:
```
# Use the best parameters to create a new pipeline
best_gbc_model = random_search_gbc.best_estimator_.named_steps['classifier']

best_gbc_pipeline = Pipeline([
    ('preprocess', CT),  # Preprocessing steps remain the same
    ('classifier', best_gbc_model)  # Replace with the new gradient boosting c
])

# Fit the pipeline to the training data
best_gbc_pipeline.fit(X_train, y_train)

# Predict the labels for the training data using the best model
y_train_pred_best_gbc = best_gbc_pipeline.predict(X_train)

# Compute the precision of the best model
precision_best_gbc = precision_score(y_train, y_train_pred_best_gbc)

print(f"Best GBC Model Precision: {precision_best_gbc}")
```

```
Best GBC Model Precision: 0.9857270294380018
```

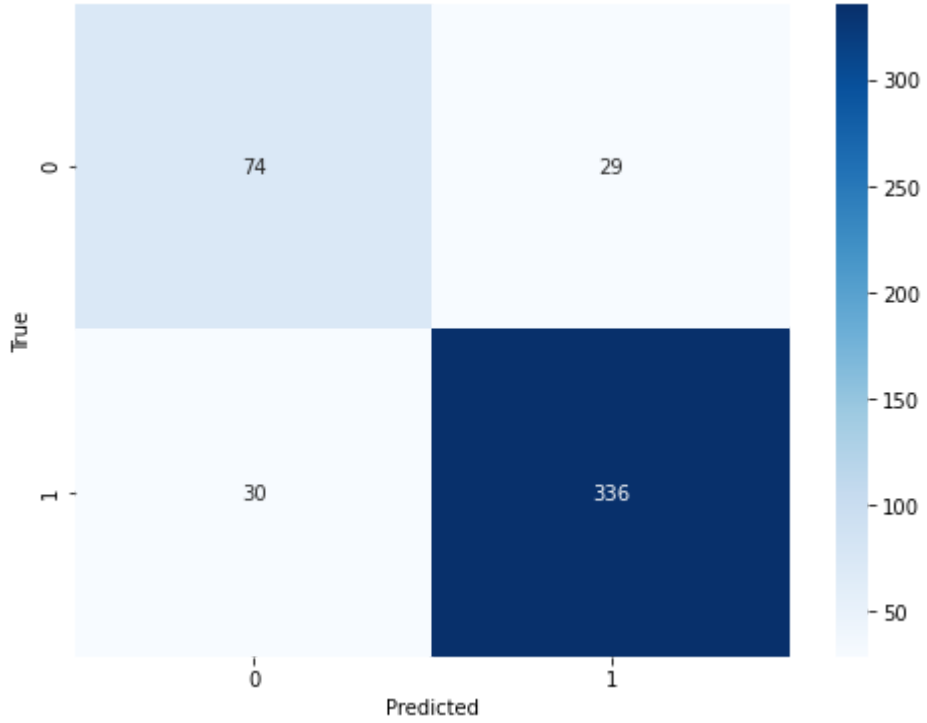
It looks like that tuning gave us a 1.5% boost. Not bad. But how does our model do with the test data?

In [36]:
```
# Predict the labels for the test data
y_test_pred_gbc = best_gbc_pipeline.predict(X_test)

# Compute the precision score for the Gradient Boosting Classifier
precision_gbc = precision_score(y_test, y_test_pred_gbc)

# Print the classification report for test data
print(classification_report(y_test, y_test_pred_gbc))
```

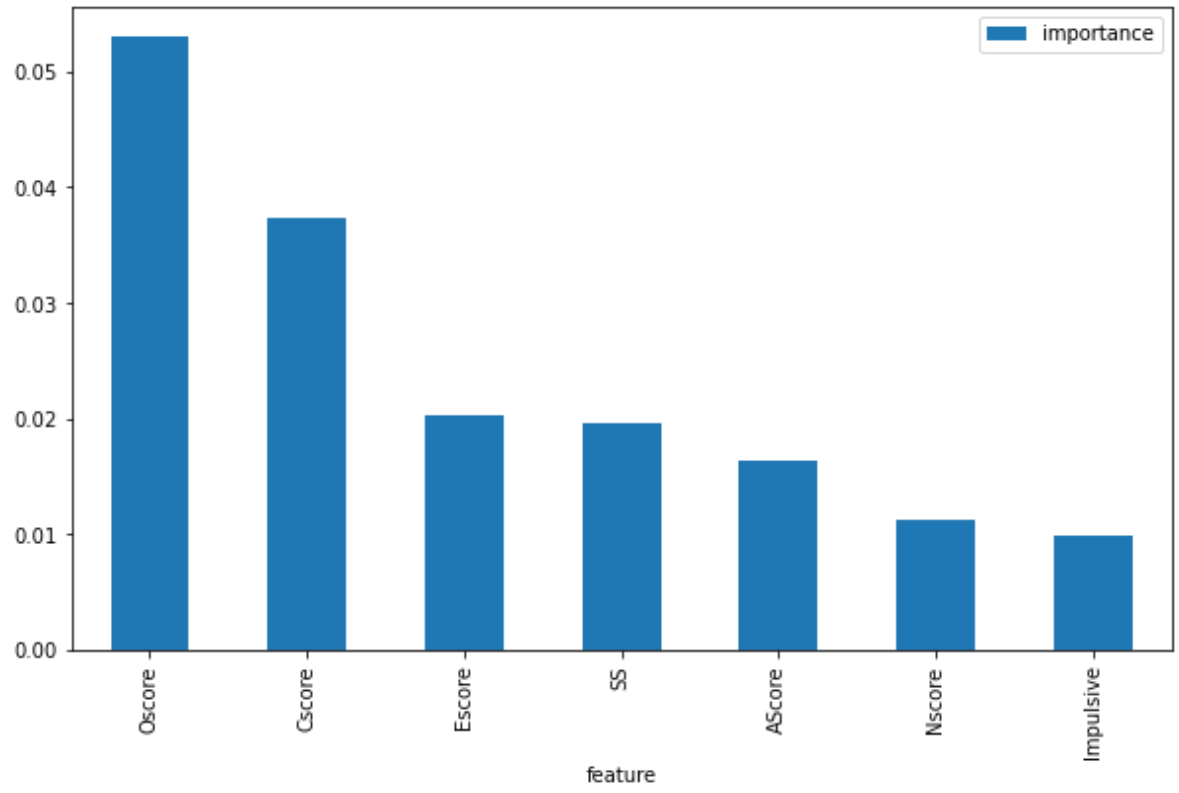
```
              precision    recall  f1-score   support

           0       0.71      0.72      0.71       103
           1       0.92      0.92      0.92       366

    accuracy                           0.87       469
   macro avg       0.82      0.82      0.82       469
weighted avg       0.87      0.87      0.87       469
```

Looks like our GBC model was overfitting. It scored a 92 for precision with our test data. A pretty big drop. And the confusion matrix indicates 29 false positives.

In [37]:
```
# Generate the predictions for the test set
y_test_pred_gbc = best_gbc_pipeline.predict(X_test)

# Generate the confusion matrix
cm = confusion_matrix(y_test, y_test_pred_gbc)

# Display the confusion matrix
plt.figure(figsize=(8,6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.show()
```



In [52]:
```
# Get column names after OneHotEncoding
cat_cols_transformed = best_gbc_pipeline.named_steps['preprocess'].named_trans

# Concatenate with numerical column names to get the final order
feature_names_ordered = np.concatenate([X_train.columns[5:12], cat_cols_transf

# Get feature importances from the Gradient Boosting Classifier in the final p
importances = best_gbc_pipeline.named_steps['classifier'].feature_importances_

# Check if the lengths of feature names and importances match
assert len(importances) == len(feature_names_ordered), "Lengths of feature nam

# Create a DataFrame with feature names and their importance
importances_df = pd.DataFrame({'feature': feature_names_ordered, 'importance':

# Sort DataFrame by importance
importances_df = importances_df.sort_values(by='importance', ascending=False)

# Filter DataFrame to include only desired features
num_features_df_gbc = importances_df[importances_df['feature'].isin(X_train.co

# Plot the feature importances for the numerical features
num_features_df_gbc.plot(kind='bar', x='feature', y='importance', figsize=(10,
```



Like our Log and RFC models, the GBC things Oscore is the most important and that Impulsive is least.

Model Comparison

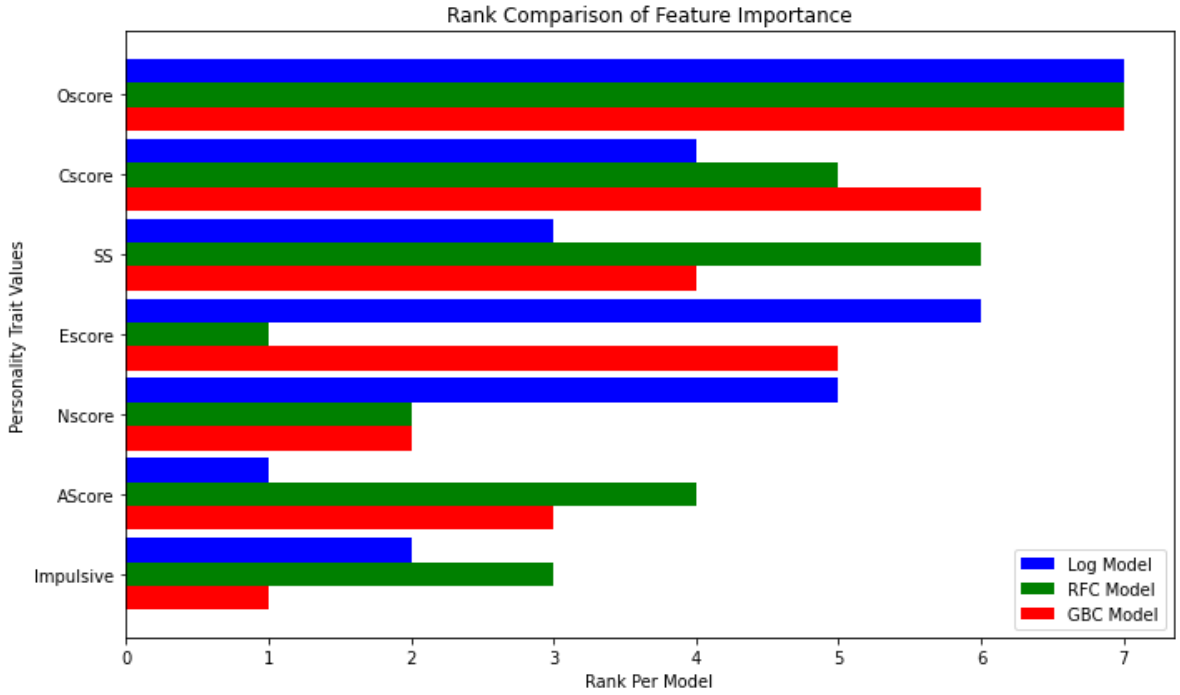
In [39]:
```
# Define the models and their predictions
models = [('Logistic Regression', log_pipeline), ('RFC', best_rfc_pipeline), (

# Define the evaluation metrics
metrics = [('accuracy', accuracy_score), ('precision', precision_score),
           ('recall', recall_score), ('F1-score', f1_score)]

# Iterate over models and metrics to compute and print the scores
for model_name, model in models:
    predictions = model.predict(X_test)
    print(f"\nModel: {model_name}")
    for metric_name, metric_func in metrics:
        score = metric_func(y_test, predictions)
        print(f"{metric_name}: {score}")
```

```
Model: Logistic Regression
accuracy: 0.8976545842217484
precision: 0.9368131868131868
recall: 0.9316939890710383
F1-score: 0.9342465753424658

Model: RFC
accuracy: 0.8848614072494669
precision: 0.9105263157894737
recall: 0.9453551912568307
F1-score: 0.9276139410187668

Model: GBC
accuracy: 0.8742004264392325
precision: 0.9205479452054794
recall: 0.9180327868852459
F1-score: 0.9192886456908346
```

When comparing all our models, it looks like our Logistical Regression model scores highest on precision, accuracy, and F1. While the scores are close, we'll give the Log model the edge and choose it to draw understandings. And if we compare feature importances focused on personality scores?

In [93]:
```
def rank_dataframe(df, column_name):
    ranked_df = df.copy()
    # Compute ranks
    ranked_df['rank'] = ranked_df[column_name].rank(ascending=False)
    # Invert ranks
    max_rank = ranked_df['rank'].max()
    ranked_df['rank'] = max_rank + 1 - ranked_df['rank']
    ranked_df = ranked_df.sort_values(by='feature')
    return ranked_df

# Create ranked dataframes using the function
num_features_df_log_ranked = rank_dataframe(num_features_df_log, 'abs_coeffici
num_features_df_rfc_ranked = rank_dataframe(num_features_df_rfc, 'importance')
num_features_df_gbc_ranked = rank_dataframe(num_features_df_gbc, 'importance')

# Calculate the average rank for each feature
average_rank = (num_features_df_log_ranked['rank'] + num_features_df_rfc_ranke

# Create a new dataframe with the average ranks
average_rank_df = pd.DataFrame({'feature': num_features_df_log_ranked['feature

# Sort the average_rank_df by the average rank, in descending order
average_rank_df = average_rank_df.sort_values(by='average_rank', ascending=Fal

# Get the order of the features based on the sorted average_rank_df
order = average_rank_df['feature']

# Set 'feature' as the index for the dataframes, so you can reorder the rows t
num_features_df_log_ranked.set_index('feature', inplace=True)
num_features_df_rfc_ranked.set_index('feature', inplace=True)
num_features_df_gbc_ranked.set_index('feature', inplace=True)

# Reorder the rows in the dataframes to match the order in average_rank_df
num_features_df_log_ranked = num_features_df_log_ranked.loc[order]
num_features_df_rfc_ranked = num_features_df_rfc_ranked.loc[order]
num_features_df_gbc_ranked = num_features_df_gbc_ranked.loc[order]

# Reset the index for the dataframes, so 'feature' is a column again
num_features_df_log_ranked.reset_index(inplace=True)
num_features_df_rfc_ranked.reset_index(inplace=True)
num_features_df_gbc_ranked.reset_index(inplace=True)

def plot_ranked_data(df1, df2, df3, label1, label2, label3):
    features = df1['feature']
    rank1 = df1['rank']
    rank2 = df2['rank']
    rank3 = df3['rank']

    fig, ax = plt.subplots(figsize=(10, 6))
    width = 0.3

    ax.barh(np.arange(len(features)), rank1, width, label=label1, color='b')
    ax.barh(np.arange(len(features)) + width, rank2, width, label=label2, colo
    ax.barh(np.arange(len(features)) + 2*width, rank3, width, label=label3, co

    ax.set_xlabel('Rank Per Model')
    ax.set_ylabel('Personality Trait Values')
```

```
    ax.set_title('Rank Comparison of Feature Importance')
    ax.set_yticks(np.arange(len(features))[::-1] + width)  # Invert the y-axis
    ax.set_yticklabels(features[::-1])  # Reverse the order of feature labels
    ax.legend()

    plt.tight_layout()
    plt.gca().invert_yaxis()  # Invert the y-axis

    plt.show()

# Call the function to plot the data
plot_ranked_data(num_features_df_log_ranked, num_features_df_rfc_ranked, num_f
```



This chart provides a comparison of feature importance rankings across the three different models: 'Log Model', 'RFC Model', and 'GBC Model'. Each horizontal bar represents a feature used in the models. The bar's length and the corresponding rank number represent the relative importance of each feature according to the specific model. The higher the rank number, the higher the importance.

Features are ordered in descending order based on the average ranking score of all three models, with the feature having the highest average rank (i.e., most often identified as important across all models) at the top.

All three models agree that Oscore is the most important, and on average, find "Impulsivness" the least important (though, for our Log Model, Ascore is actually the least important).|

Step 4 - Data Understanding

Digging Deeper into our Logistical Regression Model

First, let's get the coefficient values in our model. We probably don't need them all, but let's check for about 30 to see where our personality scores end up compared to all the other variables in terms of importance.

In [ ]:
```
# Fit the pipeline to the training data
log_pipeline.fit(X_train, y_train)

# Get the logistic regression model from the pipeline
logistic_regression_model = log_pipeline.named_steps['classifier']

# Get the coefficient values from the logistic regression model
coefficients = logistic_regression_model.coef_[0]

# Get column names after preprocessing
cat_cols_transformed = log_pipeline.named_steps['preprocess'].named_transforme
feature_names_ordered = np.concatenate([X_train.columns[5:12], cat_cols_transf

# Create a DataFrame with feature names and their absolute coefficients
coefficients_df = pd.DataFrame({'feature': feature_names_ordered, 'coefficient

# Sort the DataFrame by coefficient magnitude in descending order
coefficients_df = coefficients_df.sort_values(by='coefficient', ascending=Fals

# Display the most influential features
print(coefficients_df.head(30))
```

Out of the 130 coefficients, our Oscore is in the top 30, but there is a signifigant difference between it and our leading coefficients: Never Having Taken a Legal Highs, Nicotine, and Amyl Nitrites. A question to ask is, do people who take psychedelics and have NEVER taken a Legal High score higher on openness than non-psychedelic consumers?

In [ ]:
```
# Filter the dataset for individuals who scored 'CL0' in the 'Legalh' column
cl0_data = df_p[df_p['Legalh'] == 'CL0']

# Calculate the average 'Oscore' for individuals who scored 'CL0' in the 'Lega
average_oscore_cl0_psychedelics = cl0_data[cl0_data['Psychedelics'] == 1]['Osc

# Calculate the average 'Oscore' for individuals who scored 'CL0' in the 'Lega
average_oscore_cl0_non_psychedelics = cl0_data[cl0_data['Psychedelics'] == 0][

# Calculate the average 'Oscore' for all other results in the 'Legalh' column
average_oscore_other = df_p[df_p['Legalh'] != 'CL0']['Oscore'].mean()

# Print the results
print(f"The average Oscore for individuals who scored 'CL0' in the Legalh colu
print(f"The average Oscore for individuals who scored 'CL0' in the Legalh colu
print(f"The average Oscore for all other results in the Legalh column is: {ave
```

Looks like psychedelic users DO score higher on the Oscore than non-psychedelic users! So that indicates that Oscore possibly contributes positively to Psychedelic use in conjunction with our most important coefficient. But what about Oscore more generally?

In [ ]:
```
# Calculate the average 'Oscore' for individuals who take psychedelics
average_oscore_psychedelics = df_p[df_p['Psychedelics'] == 1]['Oscore'].mean()

# Calculate the average 'Oscore' for individuals who do not take psychedelics
average_oscore_non_psychedelics = df_p[df_p['Psychedelics'] == 0]['Oscore'].me

# Print the results
print(f"The average 'Oscore' for individuals who take psychedelics is: {averag
print(f"The average 'Oscore' for individuals who do not take psychedelics is:

# Plot 1: Bar Plot
plt.figure(figsize=(8, 6))
ax = sns.countplot(x=df_p.iloc[:, 6], hue=y, data=df_p)
num_ticks = 6
xticks = ax.get_xticks()
xtick_labels = ax.get_xticklabels()
step_size = max(1, int(len(xticks) / num_ticks))
selected_ticks = xticks[::step_size]
selected_labels = [int(float(xtick_labels[i].get_text())) for i in range(0, le
ax.set_xticks(selected_ticks)
ax.set_xticklabels(selected_labels)
plt.xlabel('Oscore')
plt.ylabel('Count')
plt.title('Oscores of Psychdelic and Nonpsychedelic Users')
plt.legend(title='Psychedelic Users')
plt.show()
```

Conclusions and Recommendations

Our first recommedation involves Recruitment Strategy. The precision of 94% achieved by the logistic regression model indicates that the model is effective in identifying potential trial participants who are genuinely likely to experiment with psychedelics. The institute can focus on targeting individuals who exhibit characteristics associated with high precision, such as never having taken legal highs, nicotine, or amyl nitrites. These factors can be used as screening criteria during the recruitment process.

Our second recommendation involves the importance of the Oscore. The Oscore coefficient with a magnitude of 0.5 indicates that it is one of the significant predictors of psychedelic use. Individuals with higher Oscores tend to be more inclined towards using psychedelics. Therefore, considering an individual's Oscore can contribute positively to the prediction of psychedelic usage. The institute can incorporate the assessment of Oscore into the screening process to further refine the selection of potential participants.

Our final recommendation involves a comparison of Oscore and psychedelic use. The analysis of the average Oscore for individuals who take psychedelics and those who do not reveals a notable difference. Individuals who take psychedelics have an average Oscore of 0.152, while those who do not have an average Oscore of -0.593. This indicates that Oscore may be a relevant factor in understanding the inclination towards psychedelic use. The institute can explore further research to investigate the relationship between Oscore and the therapeutic effects of psychedelic-assisted therapies.