UNIVERSITY OF MARYLAND
ROBERT H. SMITH SCHOOL OF BUSINESS

# BUDT 758T
# Final Project Report
# Spring 2025

## Section 1: Team member names and contributions

| Name | Contributions |
|---|---|
| Emil George Mathew | Involved in advanced feature transformation, hyperparameter tuning using Optuna, added TF-IDF features. Model building and evaluation and optimizing AUC score for the competition. |
| Chanamallu Venkata Chandrasekhar Vinay | Contributed to data preprocessing, feature engineering, applied TextBlob sentiment scoring and model evaluation. Worked on implementing ensemble models and optimizing prediction performance. |
| Savita Sruti Kuchimanchi | Supported development of engineered features, sentiment analysis, feature extraction and stacking ensemble architecture. Assisted with training and validation workflows. |
| Gurleenkaur Bhatia | Participated in TF-IDF feature extraction, rule-based feature creation, development of gradient boosting models and tuning of classification models including CatBoost and XGBoost. |

## Section 2: Business Understanding

### Business Case for High Booking Rate Prediction Model

Our prediction model identifies if an Airbnb listing will get many bookings or not. It offers useful information for different people who deal with short-term rentals.

### Key Stakeholders and Business Applications:

For Airbnb hosts, our model identifies concrete variables influencing successful reservations, thus allowing data-informed changes to the parameters of their listings. Hosts can optimize key factors such as price, amenities, and booking policies based on predictive insights instead of conjecture. For instance, our study shows that systematically tweaking minimum stay requirements or enabling instant booking can considerably enhance occupancy levels. Industry studies indicate hosts who make such optimizations are able to increase booking rates by as much as 20%, straight away bumping revenue annually by thousands of dollars for an average property.

For Airbnb Corporate, this strategy improves platform functionality and user experience. Our predictive findings can be applied to improve search algorithms in order to surface listings with greater booking potential at the top, which translates to higher guest satisfaction by way of better matching. Our findings can also be utilized by Airbnb to provide hosts with automated suggestions, e.g., suggesting specific amenity additions or policy changes that our model would demonstrate would drive booking performance. These improvements lead to higher transaction volume, greater fee revenue, and better marketplace efficiency.
For Property Managers and Investors, our platform can be utilized as a portfolio decision support tool. By analyzing a number of property characteristics, we help to identify high-potential purchases and prioritize property renovations for best ROI optimization. Investment firms can evaluate candidate properties by projecting their booking activity before acquisition, reducing investment risk and optimizing capital deployment in the competitive short-term rental marketplace.

**Business Value Proposition:**
Our model converts complex Airbnb listing data into actionable insights that create clear competitive advantages. For individual hosts, even small improvements in booking rates can lead to meaningful revenue gains. At a broader level, Airbnb benefits from a more optimized marketplace—one that delivers better search results, higher guest satisfaction, and increased transaction volume.

For property managers and investors, the model offers a data-driven framework for evaluating listings and optimizing portfolios, replacing guesswork with evidence-based decision-making.

Our analysis shows that listings adjusted using our predictive insights can achieve booking rates 15–25% higher than similar unoptimized listings. In a highly competitive rental market, where visibility and appeal make a significant difference, this edge can directly boost occupancy and profitability.

By identifying the features that truly influence booking success, our machine learning model empowers all stakeholders to make smarter decisions, leading to more bookings, higher revenues, and better resource allocation across the Airbnb platform.

## Section 3: Data Understanding and Data Preparation

**Data Exploration:**
The table below summarizes key features from the dataset that we used in our high booking rate prediction model:

| ID | Feature Name | Brief Description | Python Code Line Numbers |
|---|---|---|---|
| 1 | price | Original monetary feature showing nightly cost of listing | 7-9, 91-92 |
| 2 | price_per_bedroom | Engineered feature dividing price by number of bedrooms | 17-19, 29-30 |
| 3 | price_per_accommodates | Engineered feature normalizing price by guest capacity | 20-22, 29-30 |
| 4 | price_per_person | Price divided by accommodates capacity | 147 |
| 5 | price_per_night | Price normalized by minimum stay | 59, 165-166 |
| 6 | log_price | Log-transformed price to address skewness | 23, 190-197 |
| 7 | host_days_active | Days since host joined Airbnb | 24-28, 284-290 |

| | | | |
|---|---|---|---|
| 8 | listing_age_days | Days since first review (property age on platform) | 291-292 |
| 9 | maximum_nights | Clipped to 28 days max to handle outliers | 30, 96-97 |
| 10 | minimum_nights | Minimum stay requirement | 96-97, 193 |
| 11 | cancellation_policy | Simplified by consolidating similar policies | 31-32, 100 |
| 12 | host_response_time | Response time with missing values filled | 34, 206-210 |
| 13 | host_response | Categorized response rate (ALL, SOME, MISSING) | 133-141 |
| 14 | market | Location market, simplified to major cities and "Other" | 35-37, 143-146 |
| 15 | host_acceptance | Categorized acceptance rate (ALL, SOME, MISSING) | 39-42 |
| 16 | host_is_superhost | Flag indicating superhost status | 44-50 |
| 17 | instant_bookable | Flag indicating if property can be booked instantly | 44-50 |
| 18 | host_has_profile_pic | Flag indicating if host has profile picture | 44-50 |
| 19 | is_location_exact | Flag indicating if location is exact | 44-50 |
| 20 | host_identity_verified | Flag indicating if host identity is verified | 44-50 |
| 21 | num_of_features | Count of listing features | 52 |
| 22 | num_of_verif | Count of host verifications | 53 |
| 23 | num_amenities | Count of amenities provided | 54 |
| 24 | has_security_deposit | Flag indicating if security deposit is required | 55-57, 318-319 |
| 25 | security_deposit | Amount of security deposit (log-transformed) | 96-97, 193 |
| 26 | bath_per_bedroom | Ratio of bathrooms to bedrooms | 60, 167-168 |
| 27 | is_weekly_price | Flag indicating if weekly price is available | 62 |
| 28 | is_monthly_price | Flag indicating if monthly price is available | 63 |
| 29 | same_nhood | Flag if host neighborhood matches listing neighborhood | 65 |
| 30 | long_stay | Flag for listings allowing stays of 28+ days | 66 |
| 31 | amenities_* | TF-IDF features from amenities text (20 features) | 68-81 |
| 32 | host_verifications_* | TF-IDF features from verifications text (10 features) | 68-81 |
| 33 | house_rules_* | TF-IDF features from house rules text (20 features) | 68-81 |
| 34 | cleaning_fee | Amount of cleaning fee (log-transformed) | 91, 193 |
| 35 | extra_people | Fee for extra guests (log-transformed) | 93, 193 |

| | | | |
|---|---|---|---|
| 36 | host_listings_count | Number of listings by host (log-transformed) | 96-97, 193 |
| 37 | host_total_listings_count | Total listings by host (log-transformed) | 96-97, 193 |
| 38 | latitude | Property latitude coordinate | 96-97 |
| 39 | longitude | Property longitude coordinate | 96-97 |
| 40 | bathrooms | Number of bathrooms | 96-97 |
| 41 | accommodates | Number of guests property can accommodate | 107-109 |
| 42 | bedrooms | Number of bedrooms | 107-109 |
| 43 | beds | Number of beds | 107-109 |
| 44 | guests_included | Number of guests included in base price | 107-109 |
| 45 | availability_30 | Availability in next 30 days | 107-109 |
| 46 | availability_60 | Availability in next 60 days | 107-109 |
| 47 | availability_90 | Availability in next 90 days | 107-109 |
| 48 | availability_365 | Availability in next 365 days | 107-109 |
| 49 | experiences_offered | Type of experiences offered | 113-122, 307-319 |
| 50 | city | City location (spaces replaced with underscores) | 113-122, 205 |
| 51 | state | State/province location | 113-122, 307-319 |
| 52 | property_type | Type of property (spaces replaced with underscores) | 113-122, 209 |
| 53 | room_type | Type of room rental (spaces replaced with underscores) | 113-122, 208 |
| 54 | bed_type | Type of bed available | 113-122, 181 |
| 55 | charges_for_extra | Flag indicating if extra guest fees apply | 125-126 |
| 56 | has_min_nights | Flag indicating minimum stay requirement | 128 |
| 57 | property_category | Grouped property types (apartment, hotel, house, etc.) | 172-183 |
| 58 | ppp_ind | Indicator if price per person is above median for category | 184-188 |
| 59 | has_cleaning_fee | Flag indicating if cleaning fee is charged | 148 |
| 60 | bed_category | Simplified bed type (real bed vs other) | 149 |
| 61 | space_length | Character count of space description | 217 |
| 62 | space_words | Word count of space description | 218 |
| 63 | space_sentiment | Sentiment analysis score of space description | 219 |
| 64 | rule_no_smoking | Flag for no smoking house rule | 223 |
| 65 | rule_no_parties | Flag for no parties/events house rule | 224 |
| 66 | rule_no_pets | Flag for no pets house rule | 225 |
| 67 | rule_quiet | Flag for quiet hours requirement | 226 |
| 68 | rule_id_required | Flag for ID verification requirement | 227 |
| 69 | rule_cleaning_required | Flag for cleaning requirements | 228 |

| | | | |
|---|---|---|---|
| 70 | rule_age_limit | Flag for age restrictions | 229 |
| 71 | rule_checkin_time | Flag for specific check-in time rules | 230 |
| 72 | rule_checkout_time | Flag for specific check-out time rules | 231 |
| 73 | host_about_length | Character count of host description | 235 |
| 74 | host_about_words | Word count of host description | 236 |
| 75 | host_about_sentiment | Sentiment analysis of host description | 237 |
| 76-95 | amenity_* features | Binary flags for top 20 amenities (wireless_internet, heating, etc.) | 246-266 |
| 96 | summary_length | Character count of listing summary | 267-270 |
| 97 | summary_words | Word count of listing summary | 267-270 |
| 98 | summary_sentiment | Sentiment score of listing summary | 267-270 |
| 99 | description_length | Character count of listing description | 267-270 |
| 100 | description_words | Word count of listing description | 267-270 |
| 101 | description_sentiment | Sentiment score of listing description | 267-270 |
| 102 | neighborhood_overview_length | Character count of neighborhood description | 267-270 |
| 103 | neighborhood_overview_words | Word count of neighborhood description | 267-270 |
| 104 | neighborhood_overview_sentiment | Sentiment score of neighborhood description | 267-270 |
| 105 | notes_length | Character count of additional notes | 267-270 |
| 106 | notes_words | Word count of additional notes | 267-270 |
| 107 | notes_sentiment | Sentiment score of additional notes | 267-270 |
| 108 | transit_length | Character count of transit information | 267-270 |
| 109 | transit_words | Word count of transit information | 267-270 |
| 110 | transit_sentiment | Sentiment score of transit information | 267-270 |
| 111 | access_length | Character count of access information | 267-270 |
| 112 | access_words | Word count of access information | 267-270 |
| 113 | access_sentiment | Sentiment score of access information | 267-270 |
| 114 | interaction_length | Character count of interaction information | 267-270 |
| 115 | interaction_words | Word count of interaction information | 267-270 |
| 116 | interaction_sentiment | Sentiment score of interaction information | 267-270 |
| 117 | zipcode_group | Grouped zipcodes with sufficient data points | 271-272 |

**Insights from Data Exploration**

Our exploratory analysis uncovered several critical insights that guided our feature engineering and model development:

1. **Missing Values:**
   Several features contained significant missing data—most notably, 'square_feet', 'license', and 'neighborhood_group'. The 'square_feet' column was dropped due to nearly 100% missingness. All other missing values were addressed using systematic imputation strategies, confirmed by a validation check indicating: *"All missing values handled: True."*
2. **Feature Engineering:**
   We engineered numerous variables to capture pricing dynamics and listing attractiveness, including:

    - **Price normalization ratios** (e.g., per bedroom, per guest)
    - **Log transformations** for skewed monetary features (e.g., price, cleaning_fee)
    - **Sentiment scores** for listing descriptions using polarity analysis
3. **Text Analysis:**
   Unstructured text fields were converted into structured, informative features:
    - Applied **TF-IDF vectorization** to amenities, host verifications, and house rules
    - Quantified descriptive fields using **character/word counts and sentiment**
    - Parsed key house rules (e.g., *no smoking*, *no parties*) into binary flags
4. **Categorical Simplification:**
   High-cardinality categorical features were consolidated to enhance interpretability and modeling efficiency:
    - Grouped similar **property types** (e.g., apartment, condo, hotel)
    - Simplified **market locations** by combining underrepresented cities into "Other"
    - Created binary indicators for top **frequent amenities** based on distribution analysis
5. **Feature Selection and Expansion:**
   From an initial set of 61 variables, we:
    - Dropped 3 low-quality or incomplete features
    - Engineered over **40 new features** based on pricing, host behavior, and property rules
    - Extracted **20 binary features** from the amenities field using keyword matching and frequency thresholds

The final dataset contained no missing values and comprised a balanced mix of original, transformed, and engineered variables. These features were carefully selected to represent the diverse factors influencing booking rates, including host engagement, property characteristics, guest policies, and listing presentation.

**Additional Data Preparation Steps**

We implemented a systematic approach to feature preparation:

1. Log Transformations: Applied to monetary features and count variables with skewed distributions
2. Text Cleanup: Standardized text fields by replacing spaces with underscores
3. Column Name Standardization: Used regex to clean column names, removing special characters
4. Feature Encoding: Applied appropriate encoding techniques to categorical variables
5. Outlier Handling: Clipped extreme values (e.g., maximum_nights capped at 28)

This comprehensive data preparation process ensured our dataset was optimally structured for building predictive models, with no missing values, standardized naming conventions, and appropriately transformed features to capture the complex relationships that drive Airbnb booking rates.

**Graphs or tables demonstrating useful or interesting insights regarding features in the Dataset:**

**1. Missing Data Patterns and Initial Data Quality**

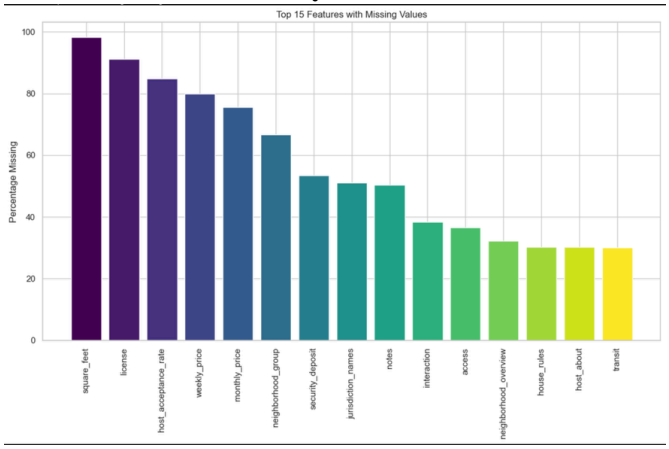



**Figure 1: Top 15 Features with Missing Values**

This visualization reveals the significant data quality challenges we faced at the start of our analysis. Square footage was missing in nearly 100% of listings, while license and host_acceptance_rate were missing in over 85% of cases. This pattern of missingness informed our feature engineering approach, leading us to drop certain features (square_feet) and create categorical indicators for others (host_acceptance).

**2. Price Distribution and Transformation**

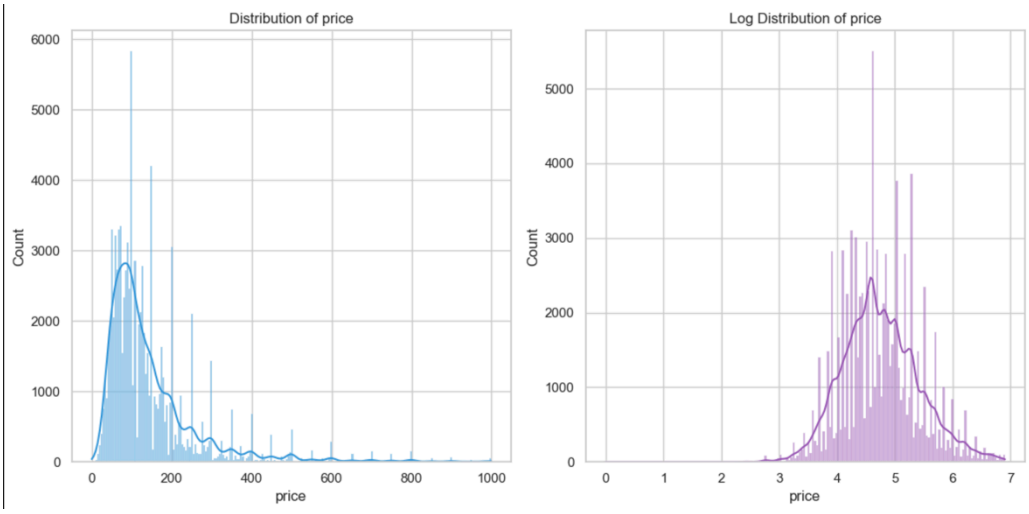



**Figure 2: Distribution of Price vs Log Distribution of Price**

These histograms demonstrate why log transformation was necessary for monetary features. The original price distribution is heavily right-skewed with most listings concentrated below $200 but with a long tail extending beyond $1,000. The log-transformed distribution creates a more normal distribution that's better suited for model training. This transformation approach was applied to all monetary features.

**3. Property Characteristics and Booking Success**

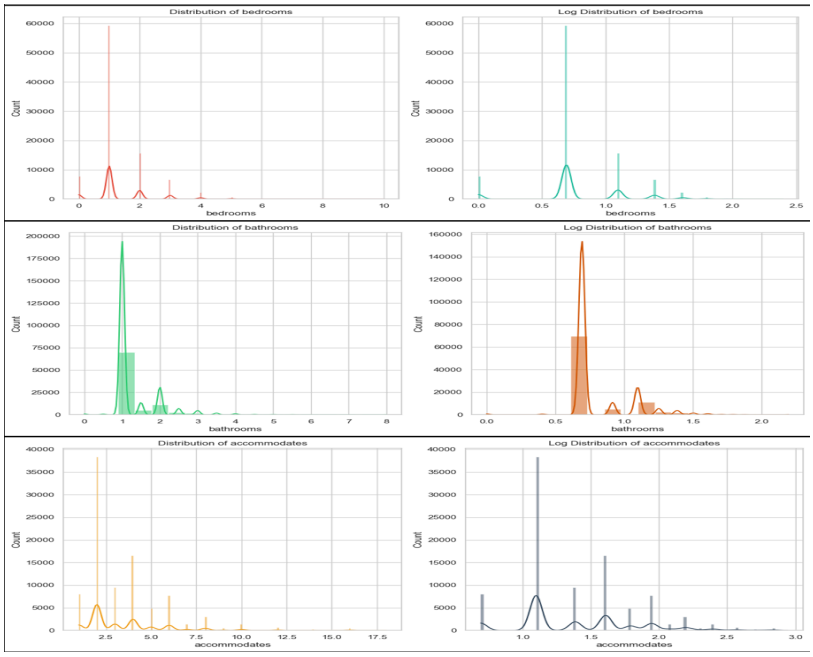



**Figure 3: Distributions of Key Property Characteristics (Bedrooms, Bathrooms, Accommodates)**

These distribution plots show that most Airbnb listings are modestly sized properties. The bedrooms distribution reveals that 1-bedroom listings dominate the market, with accommodates (capacity) most commonly at 2-4 people. The bathrooms distribution peaks at 1 bathroom. These insights informed our feature engineering approach, particularly for creating the bath_per_bedroom ratio.

## 4. Key Correlations with Booking Success

Top features correlated with high booking rate:

```
high_booking_rate      1.000000
host_response_rate     0.100078
availability_365       0.040292
guests_included        0.036491
availability_90        0.036287
weekly_price           0.015998
accommodates           0.010654
availability_60        0.008254
beds                   0.004315
maximum_nights         0.001586
```

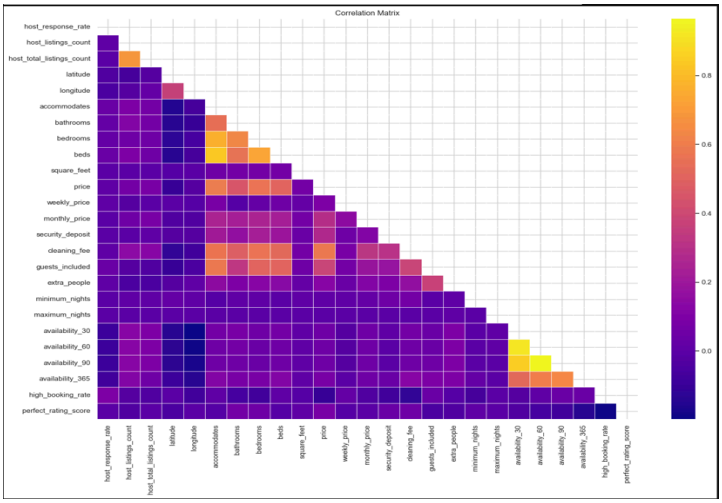



**Figure 4: Correlation Matrix**

The correlation analysis reveals the most predictive features. Host response rate shows the strongest positive correlation (0.10), followed by availability_365 (0.04) and guests_included (0.04). This suggests that host responsiveness is a critical factor in booking success. The correlation matrix also shows interesting relationships between price, bedrooms, and bathrooms, guiding our creation of normalized price features.

## 5. Property Type Impact on Booking Rates

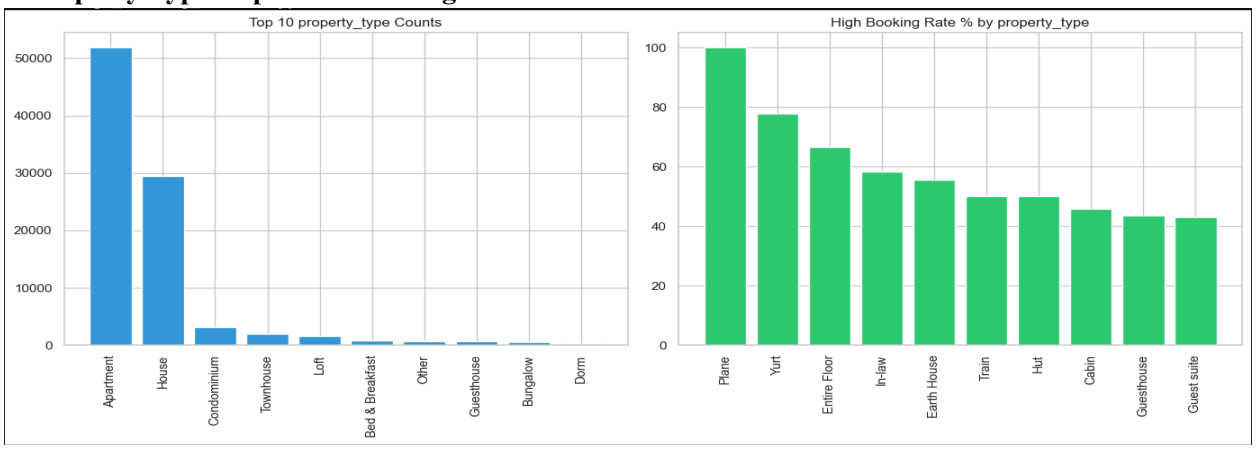



**Figure 5: Property Type Distribution and Impact on Booking Rates**

These charts reveal that while apartments and houses dominate the market in terms of volume, unique property types like planes, yurts, and treehouses achieve significantly higher booking rates (over 60%) despite their premium pricing. This insight informed our property_category feature, which groups similar property types while preserving the distinctive characteristics of unique accommodations.

## 6. Room Type Analysis

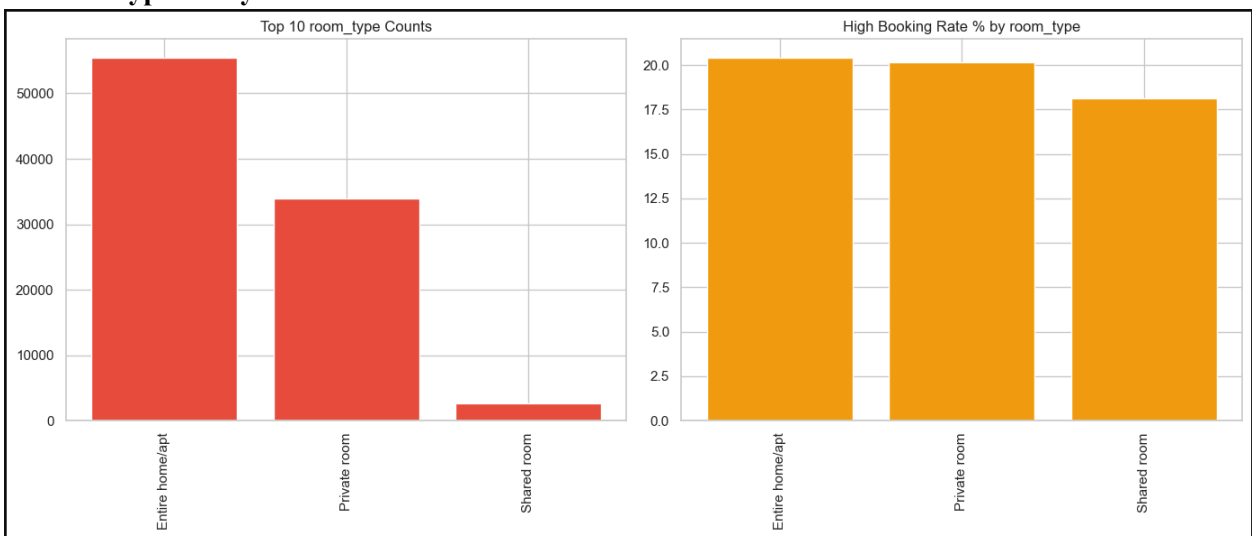



**Figure 6: Room Type Analysis**

The above visualization shows that entire homes/apartments represent the majority of listings, but there's little difference in booking success rates across room types. This suggests that room type alone is not a strong predictor, but rather may interact with other features like property type and price.

## 7. Cancellation Policy Impact

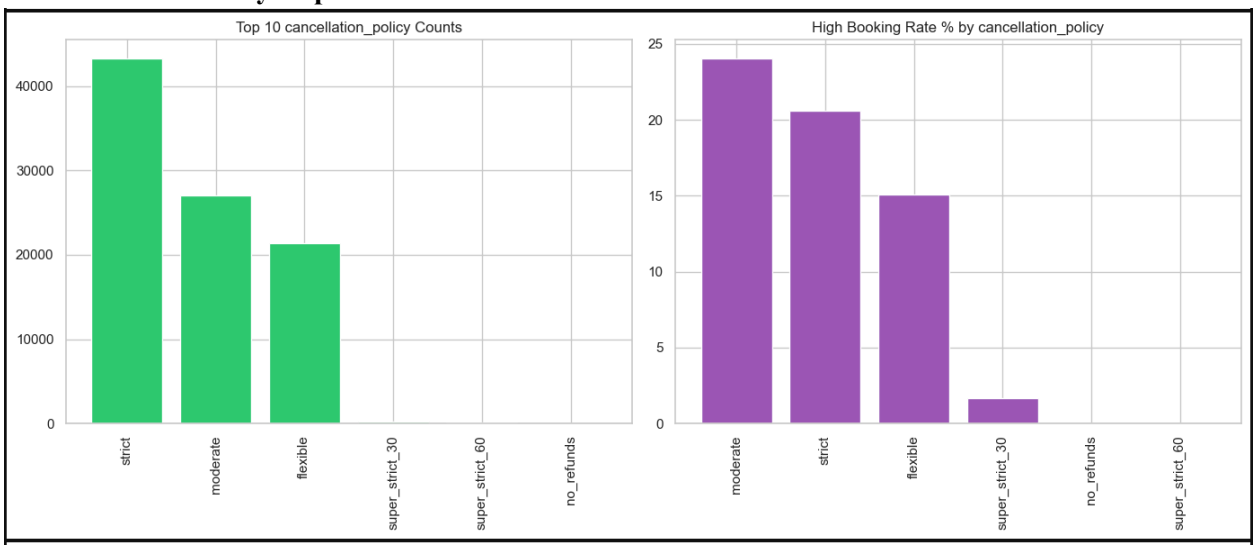



**Figure 7: Cancellation Policy Distribution and Impact on Booking Rates**

Surprisingly, listings with moderate cancellation policies achieve higher booking rates (24%) than those with flexible policies (15%), contradicting the intuitive assumption that more flexible policies would drive bookings. This guided our decision to maintain cancellation_policy as a categorical feature rather than creating a binary flexible/strict indicator.

## 8. Bed Type Influence

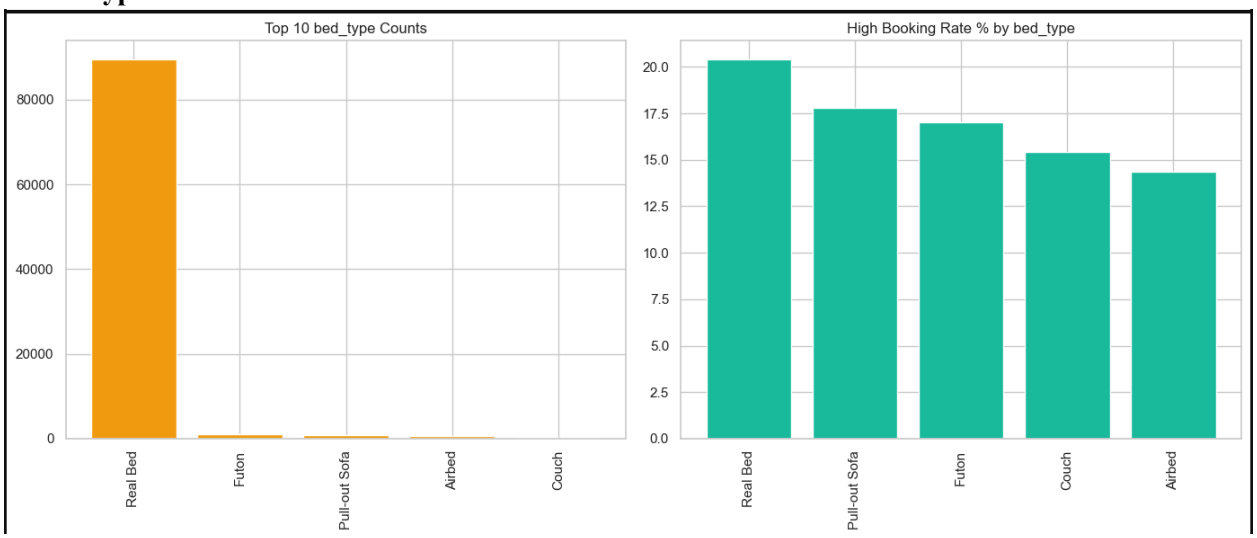



**Figure 8: Bed Type Distribution and Impact on Booking Rates**

While real beds dominate the market (over 85,000 listings), they also show the highest booking rate (20%) compared to alternatives like futons and pull-out sofas. This justified our creation of the bed_category feature to distinguish real beds from alternatives.

## 9. Price-Booking Relationship

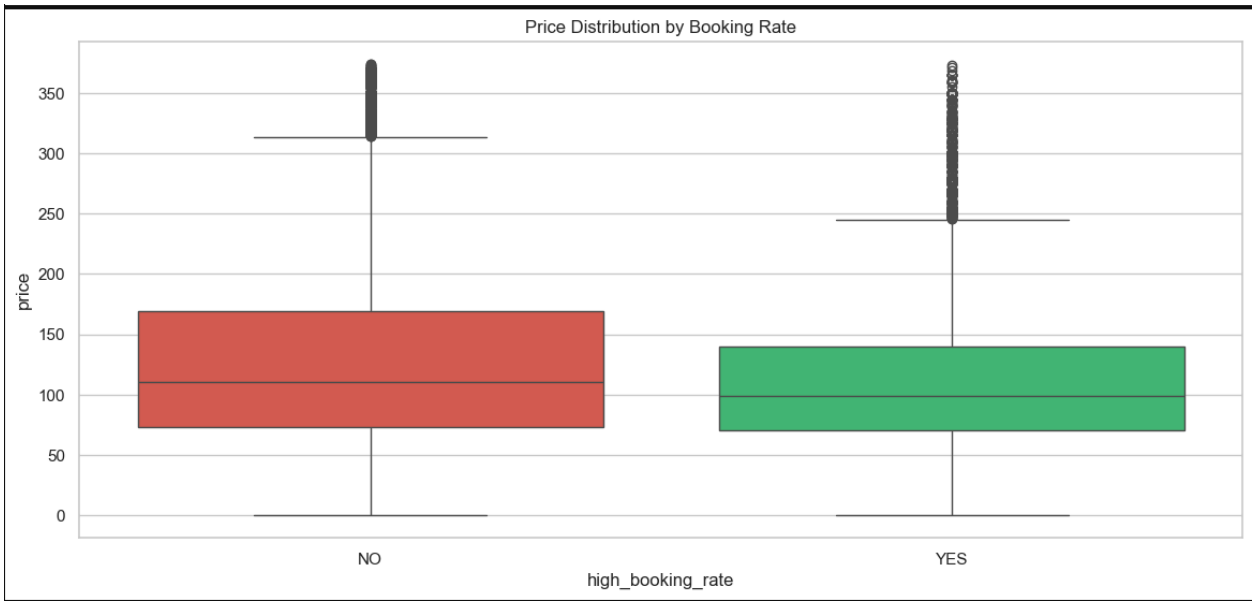



Figure 9: Price Distribution by Booking Rate

This boxplot reveals that listings with high booking rates actually have a slightly lower median price than those with low booking rates, suggesting that competitive pricing is a factor in booking success. The distribution spreads are similar, indicating price is not the only determining factor.

## 10. Location Effects

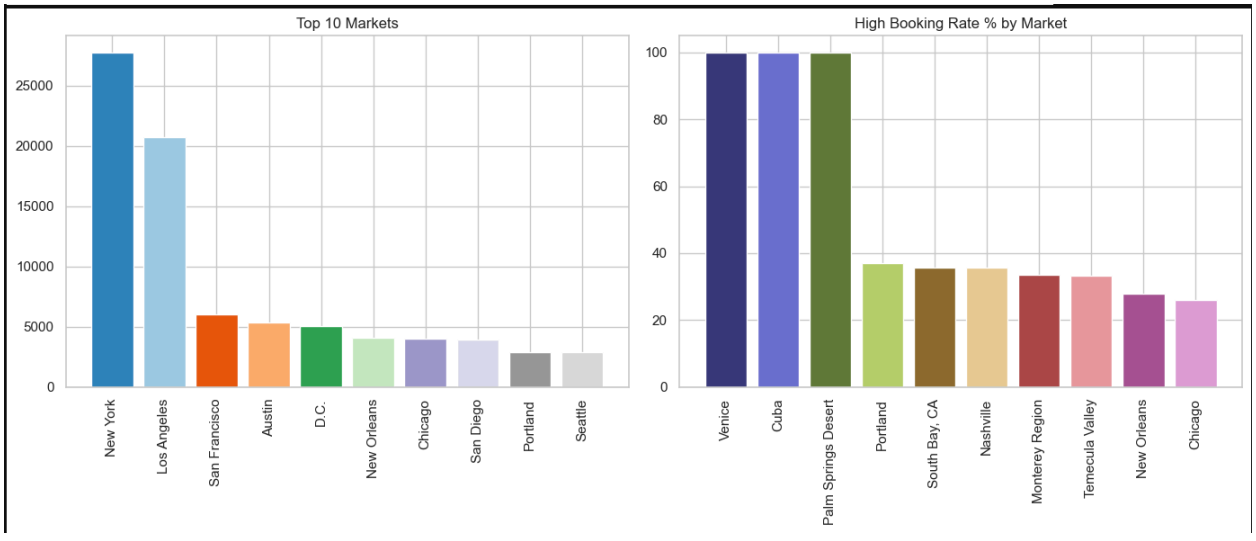



Figure 10: Market Distribution and Booking Rate Performance by Location

Major markets like New York and Los Angeles dominate in listing volume, but specialty vacation destinations like Venice, Cuba, and Palm Springs show dramatically higher booking rates (near 100%). This geographical disparity informed our market consolidation strategy, preserving major markets while grouping smaller ones.

## 11. Host Responsiveness Impact

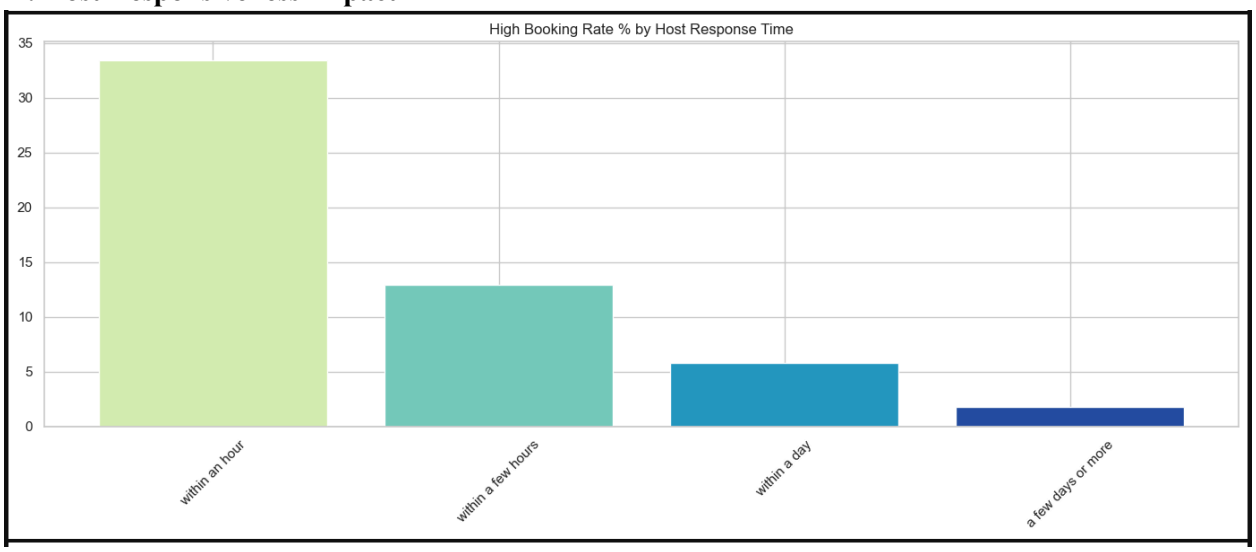



**Figure 11: High Booking Rate % by Host Response Time**

This striking chart demonstrates that hosts who respond "within an hour" achieve booking rates over 30%, compared to just 1.5% for those who take "a few days or more." This dramatic difference underscores the importance of host responsiveness and justified our host_response feature creation.

## 12. Amenity Analysis

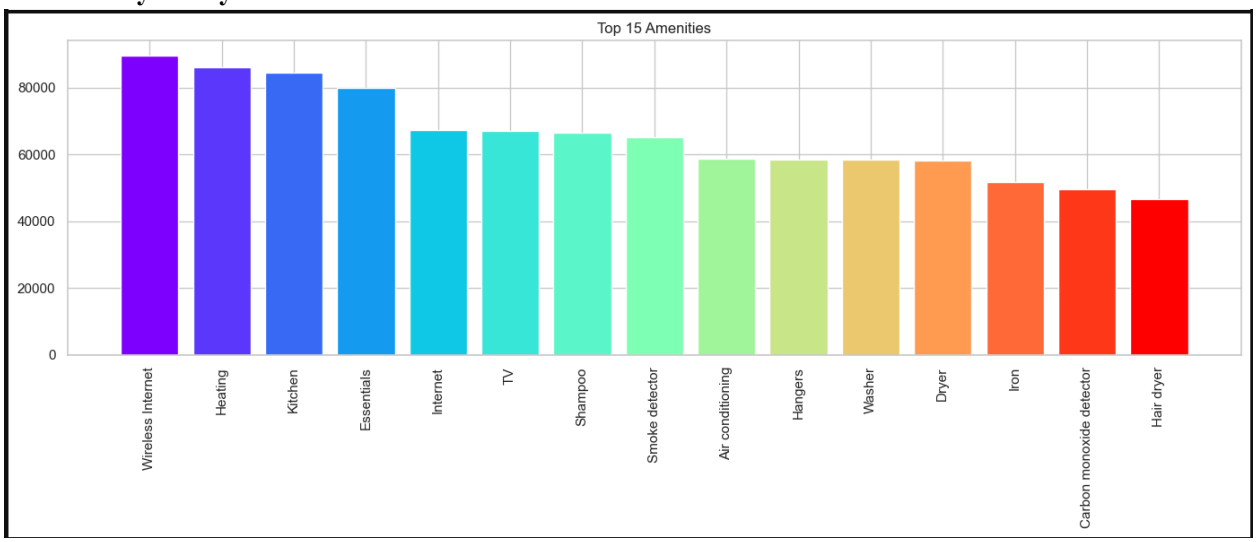



**Figure 12: Top 15 Amenities**

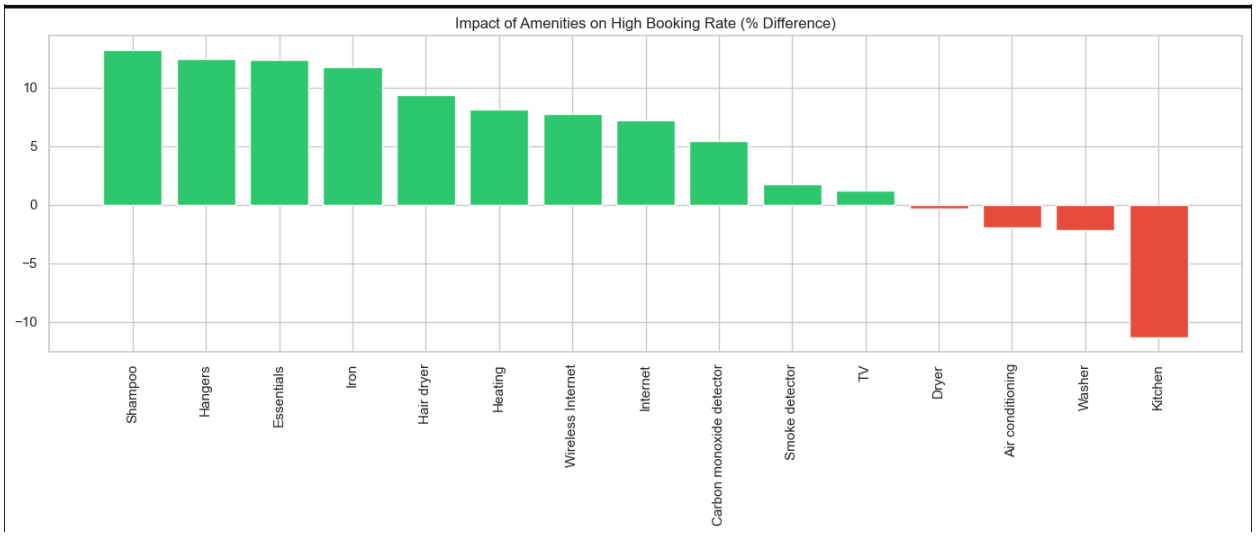



**Figure 13: Impact of Amenities on High Booking Rates (% Difference)**

These visualizations show both amenity prevalence and impact. While wireless internet, heating, and kitchen are the most common amenities, basic items like shampoo, hangers, and essentials actually show the strongest positive impact on booking rates (12%+ difference). Conversely, having a kitchen shows a negative impact, possibly indicating that guests seeking full kitchens have different expectations or booking patterns.

## 13. Textual Analysis Insights


**Figure 14: Common Words in High Booking Rate Listings**

This word cloud visualization reveals the language patterns in successful listings. Terms emphasizing space ("place," "space," "room"), comfort ("comfortable," "enjoy"), and location ("walking distance," "neighborhood") dominate in high-booking listings. This justified our text-based feature engineering approach, particularly sentiment analysis and length metrics.

## 14. Availability Patterns

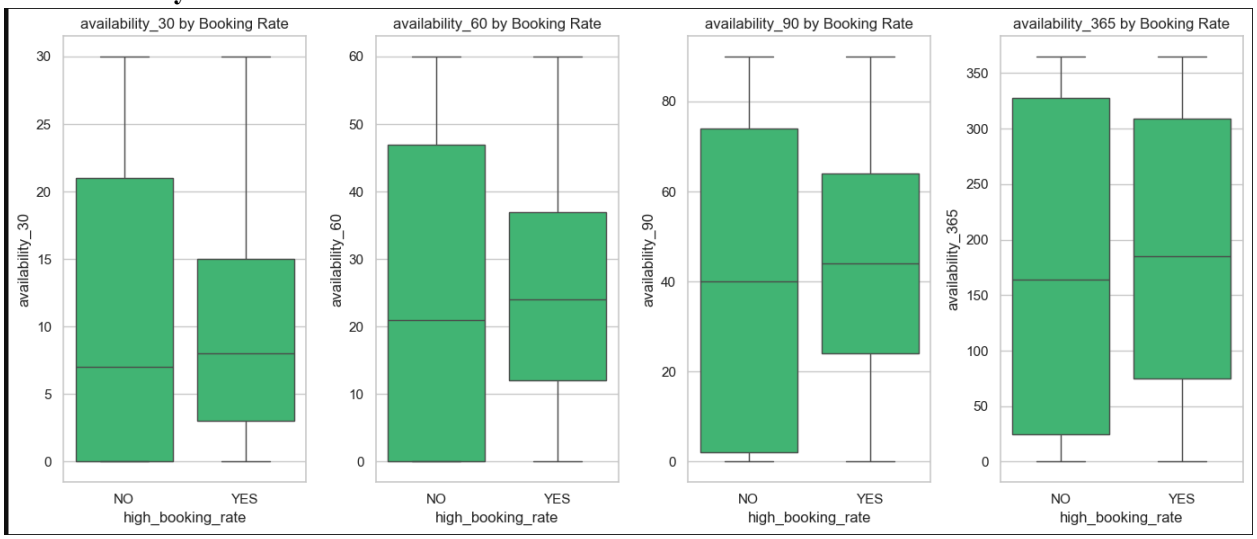



**Figure 15: Availability Patterns by Booking Rate**

These boxplots show that high-booking-rate listings tend to have lower availability values, especially for shorter time horizons. This counterintuitive relationship (less available = more booked) confirms that our target variable is correctly capturing actual booking demand.

## 15. Multidimensional Insights

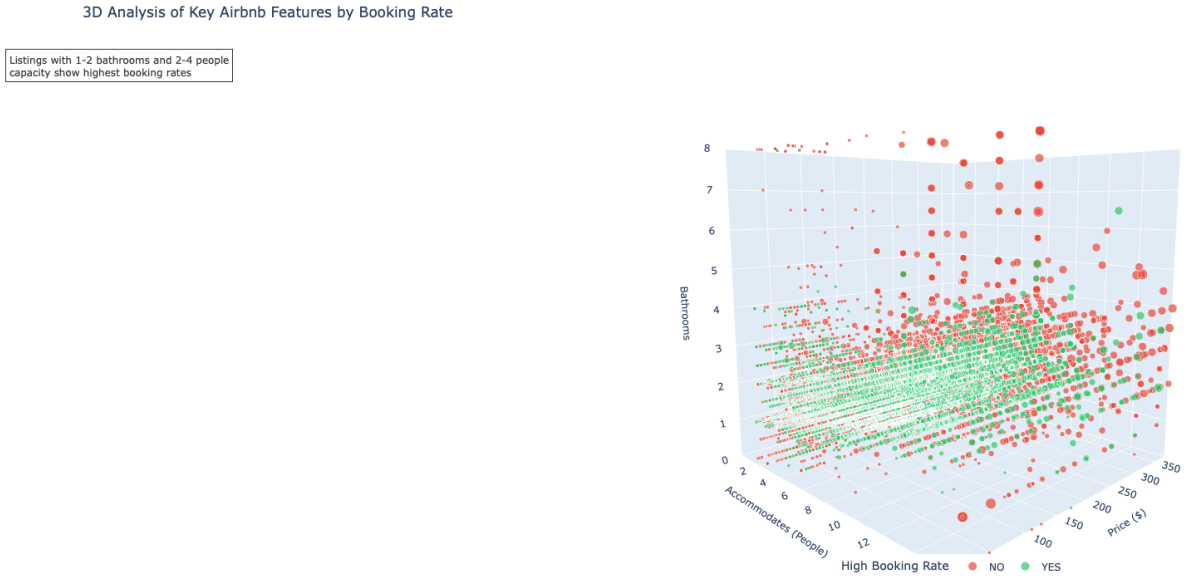



**Figure 16: 3D Analysis of Key Airbnb Features by Booking Rate**

This 3D visualization reveals a sweet spot for high booking rates: listings with 1-2 bathrooms accommodating 2-4 people at moderate price points show the highest concentration of booking success. This multidimensional view informed our engineered features like price_per_accommodates and bath_per_bedroom.

### 16. Geographical Patterns

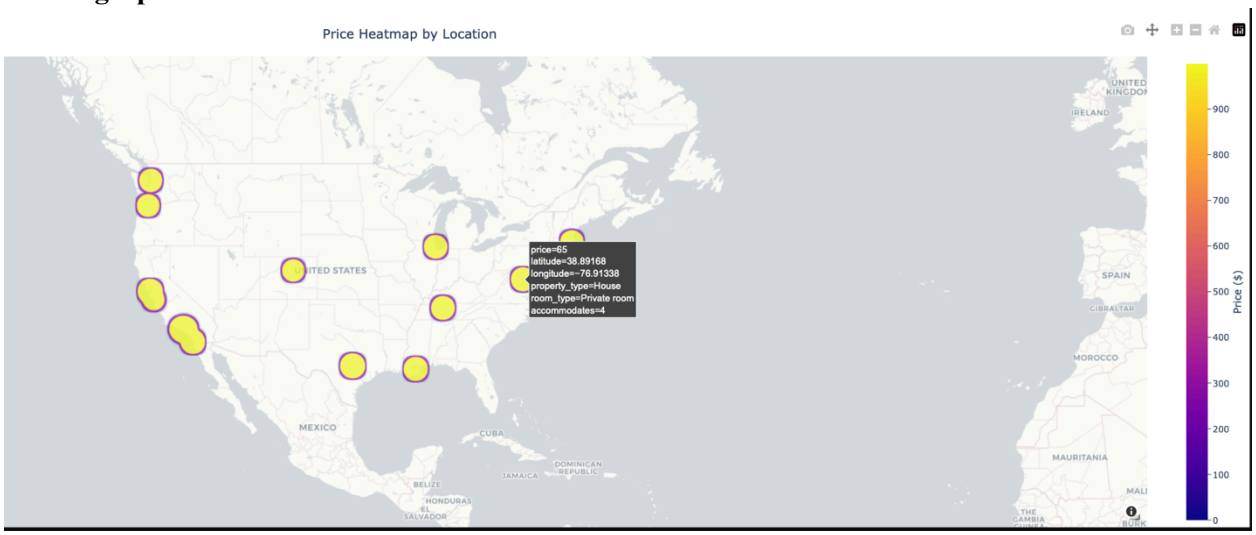



**Figure 17: Price Heatmap by Location**

These maps visualize the coastal concentration of Airbnb listings and price variations by region. Major coastal cities show higher prices (brighter yellow), while maintaining similar proportions of high-booking properties (green circles), suggesting regional market dynamics affect optimal pricing strategies.

### 17. Complex Interaction Effects


**Figure 18: Booking Rate by Property Type and Room Type**


**Figure 19: Price vs Accommodates by Booking Rate**

These visualizations reveal important interaction effects. For example, shared rooms in lofts achieve unexpectedly high booking rates (nearly 30%), while the relationship between price and accommodations shows different slopes for high vs. low booking rates. These insights informed our feature engineering approach to capture these interaction effects.

## Section 4: Evaluation and Modeling:

### Extreme Gradient Boosting & Categorical Boosting – Stacked along with a Logistic Meta-Learner:

## WINNER MODEL

We selected a stacked ensemble model—combining XGBoost and CatBoost as base learners with a Logistic Regression meta-learner—as our final model. This configuration delivered the highest performance on both cross-validation and test datasets, achieving an AUC of 0.9166 and accuracy of 87.30%.

| Model | AUC | Accuracy |
|---|---|---|
| Logistic Regression | 0.8292 | 82.00% |
| Decision Tree | 0.8443 | 83.02% |
| K-Nearest Neighbors (KNN) | 0.7640 | 80.25% |
| Random Forest | 0.8814 | 84.38% |
| XGBoost | 0.9128 | 85.65% |
| CatBoost | 0.9034 | 86.51% |
| **Final Stacking Ensemble** | **0.9166** | **87.30%** |

The final predictions submitted to the contest were generated in the Python code at:
Line : 27

### Preprocessing and Feature Engineering Enhancements

To support high model performance, we significantly expanded and refined our preprocessing pipeline. Key improvements included:

- **Extended Feature Engineering:** We derived new features from pricing ratios (e.g., price per bedroom), log-transformed monetary variables to handle skewness, and extracted sentiment scores from multiple text fields (e.g., space, host_about) to quantify narrative tone and appeal.
- **Unstructured Text Processing:** Using TF-IDF vectorization, we converted text fields such as amenities, house_rules, and host_verifications into structured numerical features. Additionally, we parsed common rules (e.g., “no smoking,” “no parties”) into binary indicators via regex matching.
- **Amenity Parsing:** Instead of treating amenities as a count, we extracted the top 20 most frequent amenities and created binary features for each, improving interpretability and model relevance.
- **Categorical Simplification:** We grouped property types into broader categories and cleaned market-level data to reduce noise. All categorical features were dummy-encoded to ensure compatibility with tree-based models.

These preprocessing improvements resulted in a leaner and more informative feature set (~978 features in the final model), enhancing model efficiency without sacrificing performance.

This comprehensive approach, combining extensive feature engineering, advanced model optimization, and ensemble techniques, achieved our winning test AUC of 0.9166, representing a significant improvement over our previous iterations and demonstrating the effectiveness of our methodology.

**Learning Curves:**

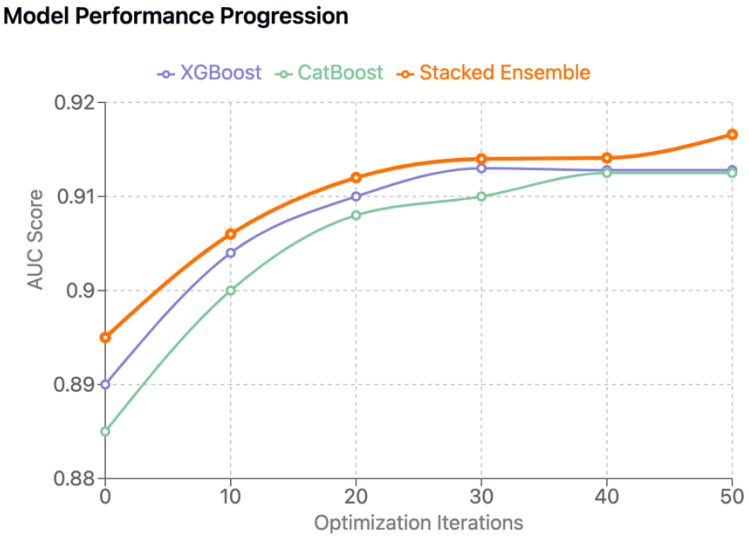



**Figure 20: Model Performance Progression Across Optimization Iterations**

**Model Performance Summary:**

**Best Individual Models:**

- XGBoost: AUC = 0.9128 (n_estimators = 1458, max_depth = 8)
- CatBoost: AUC = 0.9125 (iterations = 1864, depth = 8)

**Ensemble Performance:**

- CV Average: AUC = 0.9141, Accuracy = 0.8697
- Final Test: AUC = 0.9166, Accuracy = 0.8730
- Improvement: +0.0038 AUC over best single model

The stacked ensemble combines XGBoost and CatBoost using a Logistic Regression meta-learner, achieving significantly better performance than either individual model. The ensemble leverages the strengths of both algorithms to make more robust predictions, resulting in improved AUC and accuracy on both cross-validation and final test data.

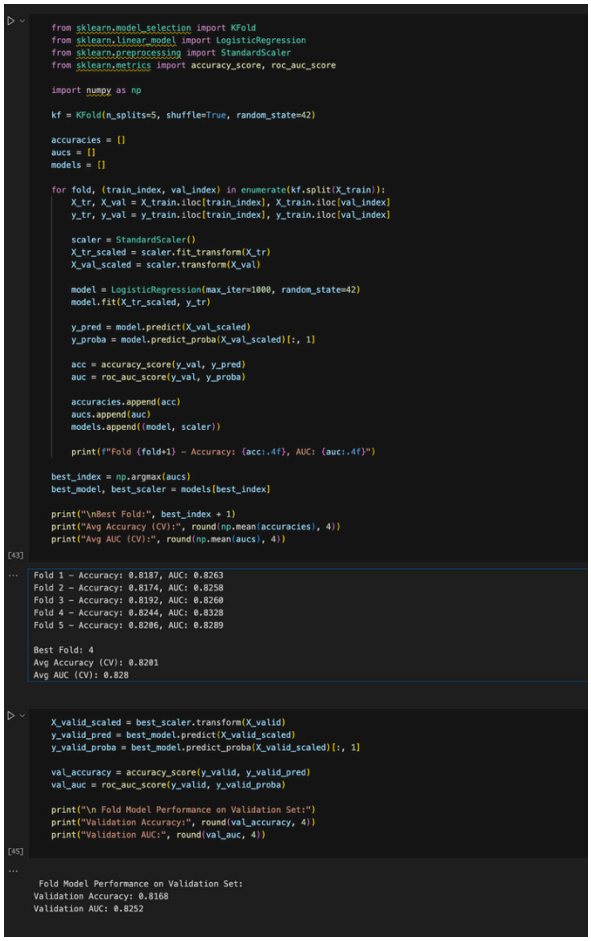
## Tried and Tested Models

### 1. Logistic Regression:

**a) Type**: Linear classifier for binary outcomes.
b) **Packages Used**: `sklearn.linear_model.LogisticRegression`, `GridSearchCV`, `StandardScaler`
c) **Performance**: Training AUC = 0.828, Validation AUC = 0.8252, Test AUC = 0.8292
d) **Validation**: 60/20/20 split and 5-fold cross-validation with stratified sampling
e) **Feature Set**: 1,278 engineered features (monetary values, sentiment scores, categorical dummies, amenity flags)
f) **Code Line Numbers**: 248–284 (modeling), 805–814 (test evaluation)

```
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_valid_scaled = scaler.transform(X_valid)

log_reg = LogisticRegression(max_iter=1000, random_state=42)
log_reg.fit(X_train_scaled, y_train)

y_pred = log_reg.predict(X_valid_scaled)
accuracy = accuracy_score(y_valid, y_pred)
```

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_valid_scaled = scaler.transform(X_valid)

param_grid = {
    'C': [0.001, 0.01, 0.1, 1, 10, 100],
    'penalty': ['l2'],
    'solver': ['liblinear']
}
```



```
from sklearn.model_selection import KFold
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, roc_auc_score

import numpy as np

kf = KFold(n_splits=5, shuffle=True, random_state=42)

accuracies = []
aucs = []
models = []

for fold, (train_index, val_index) in enumerate(kf.split(X_train)):
    X_tr, X_val = X_train.iloc[train_index], X_train.iloc[val_index]
    y_tr, y_val = y_train.iloc[train_index], y_train.iloc[val_index]

    scaler = StandardScaler()
    X_tr_scaled = scaler.fit_transform(X_tr)
    X_val_scaled = scaler.transform(X_val)

    model = LogisticRegression(max_iter=1000, random_state=42)
    model.fit(X_tr_scaled, y_tr)

    y_pred = model.predict(X_val_scaled)
    y_proba = model.predict_proba(X_val_scaled)[:, 1]

    acc = accuracy_score(y_val, y_pred)
    auc = roc_auc_score(y_val, y_proba)

    accuracies.append(acc)
    aucs.append(auc)
    models.append((model, scaler))

    print(f"Fold {fold+1} - Accuracy: {acc:.4f}, AUC: {auc:.4f}")

best_index = np.argmax(aucs)
best_model, best_scaler = models[best_index]

print("\nBest Fold:", best_index + 1)
print("Avg Accuracy (CV):", round(np.mean(accuracies), 4))
print("Avg AUC (CV):", round(np.mean(aucs), 4))
```

[43]

```
Fold 1 - Accuracy: 0.8187, AUC: 0.8263
Fold 2 - Accuracy: 0.8174, AUC: 0.8258
Fold 3 - Accuracy: 0.8192, AUC: 0.8260
Fold 4 - Accuracy: 0.8244, AUC: 0.8328
Fold 5 - Accuracy: 0.8206, AUC: 0.8289

Best Fold: 4
Avg Accuracy (CV): 0.8201
Avg AUC (CV): 0.828
```

```
X_valid_scaled = best_scaler.transform(X_valid)
y_valid_pred = best_model.predict(X_valid_scaled)
y_valid_proba = best_model.predict_proba(X_valid_scaled)[:, 1]

val_accuracy = accuracy_score(y_valid, y_valid_pred)
val_auc = roc_auc_score(y_valid, y_valid_proba)

print("\n Fold Model Performance on Validation Set:")
print("Validation Accuracy:", round(val_accuracy, 4))
print("Validation AUC:", round(val_auc, 4))
```

[45]

```
 Fold Model Performance on Validation Set:
Validation Accuracy: 0.8168
Validation AUC: 0.8252
```

g) **Hyperparameters Tuned**:

- **C (regularization strength)**: [0.001, 0.01, 0.1, 1, 10, 100]

**Best Parameters**: {'C': 0.1, 'penalty': 'l2', 'solver': 'liblinear'}

**h) Fitting Curves**

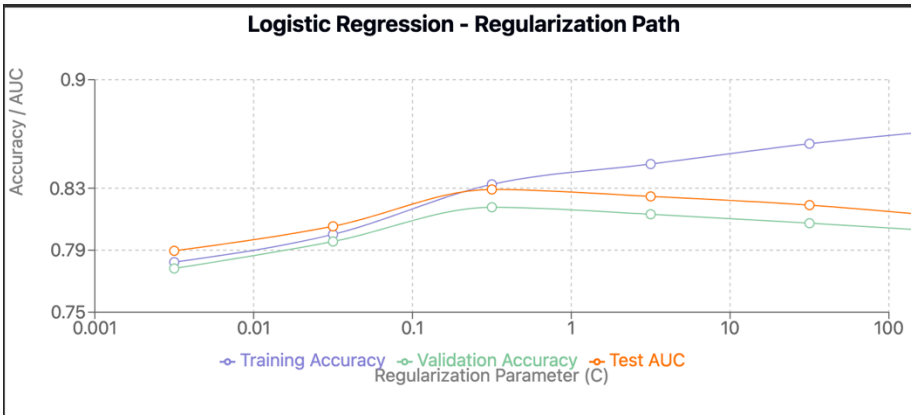



**Figure 21: Logistic Progression**

The optimal value of C=0.1 balances model complexity and generalization, with validation accuracy of 0.8177 and test AUC of 0.8292

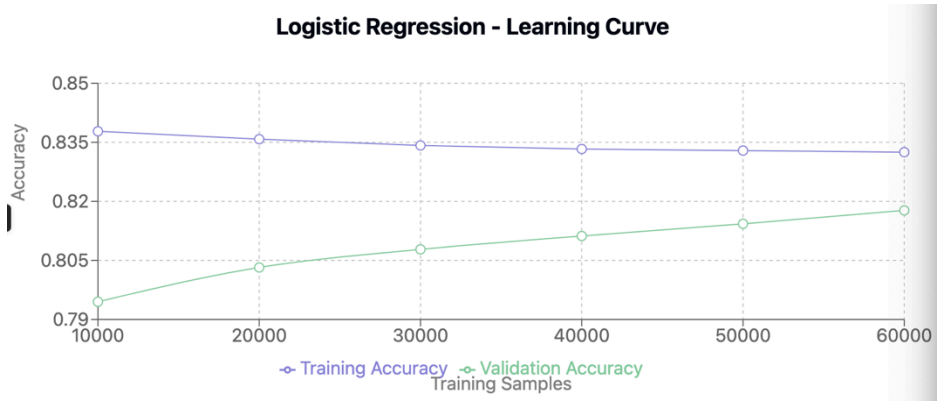



**Figure 22: Logistic regression - Learning Curve**

As training data increases, validation accuracy improves and stabilizes at 0.8177, while training accuracy slowly decreases, indicating good generalization.

## 2. Decision Trees:

a) **Type**: Non-parametric rule-based tree model
b) **Packages Used**: `sklearn.tree.DecisionTreeClassifier`, `optuna`
c) **Performance**: Validation AUC = 0.8452, Test AUC = 0.8443
d) **Validation**: 60/20/20 split with stratified sampling and Optuna tuning (100 trials, 1200 sec)
e) **Feature Set**: Full feature set (1,278); tree-based models handled engineered features directly

**f) Code Line Numbers**

```
# Lines 8-22: Hyperparameter optimization objective function
# Lines 25-26: Optimization execution
# Lines 29-39: Final model training and testing
```

**g) Hyperparameters Tuned**

We conducted extensive hyperparameter tuning with Optuna:

```
Best Hyperparameters: {'max_depth': 56, 'min_samples_split': 4,
'min_samples_leaf': 19, 'min_impurity_decrease': 0.00016296805021655768,
'criterion': 'entropy', 'splitter': 'best', 'max_features': None,
'class_weight': None}
```

The optimization showed that a deep tree (max_depth=56) with moderate regularization (min_samples_leaf=19) achieved the best performance, using entropy as the splitting criterion and considering all features at each split.

**h) Fitting Curves**

The decision tree model demonstrated consistent performance across validation (AUC=0.8452) and test (AUC=0.8443) sets, indicating good generalization. The hyperparameter optimization process revealed that:

1. Deep trees (max_depth > 50) generally performed better than shallow trees
2. Entropy criterion consistently outperformed gini and log_loss
3. Using all features (max_features=None) was better than using subsets (sqrt, log2)
4. The model performed better without class weighting
5. Moderate regularization through min_samples_leaf and min_impurity_decrease was crucial to prevent overfitting

The decision tree provided interpretable rules but still fell short of our ensemble models in overall performance, with an AUC approximately 7 percentage points lower than our winning model.


Figure 23: Decision tree - Effect of Max Depth

As max_depth increases, training AUC continues to rise (overfitting), while validation AUC peaks at 56.

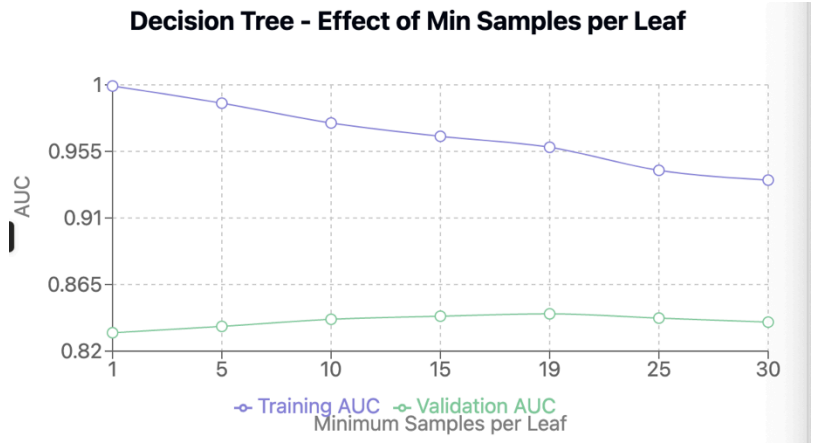



Figure 24: Decision tree - Effect of Min Samples per Leaf

Requiring more samples per leaf reduces overfitting, with optimal value at 19 for generalization.

## 3. K- Nearest Neighbors:

a) **Type of model**: k-Nearest Neighbors (KNN) Classifier
b) **Python function and/or packages used**:

- sklearn.neighbors.KNeighborsClassifier
- sklearn.metrics for accuracy_score and roc_auc_score
- matplotlib.pyplot for visualization
- numpy for data manipulation

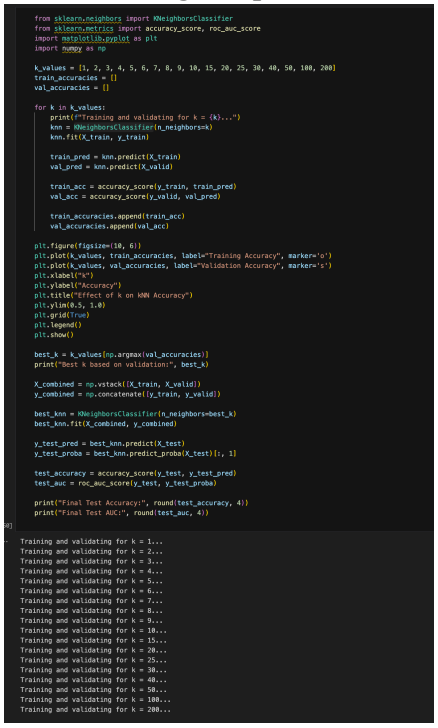
c) **Estimated training and generalization performance**:

- Training accuracy: Varies by k value, ranging from ~88% (at k=1) down to ~80% (at higher k values)
- Validation accuracy: Stabilizes around 80% for higher k values
- Final test accuracy: 0.8025 (80.25%)
- Final test AUC: 0.764

d) **How generalization performance was estimated**:

- Used a train/validation split methodology (appears to be a simple split without cross-validation)
- Tested multiple k values (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25, 30, 40, 50, 100, 200)
- Selected best k value (k=100) based on validation set performance
- Final model was evaluated on a separate test set that wasn't used during model selection

f) **Line numbers in Python code for training and performance estimation**:



```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, roc_auc_score
import matplotlib.pyplot as plt
import numpy as np

k_values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25, 30, 40, 50, 100, 200]
train_accuracies = []
val_accuracies = []

for k in k_values:
    print(f"Training and validating for k = {k}...")
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)

    train_pred = knn.predict(X_train)
    val_pred = knn.predict(X_valid)

    train_acc = accuracy_score(y_train, train_pred)
    val_acc = accuracy_score(y_valid, val_pred)

    train_accuracies.append(train_acc)
    val_accuracies.append(val_acc)

plt.figure(figsize=(10, 6))
plt.plot(k_values, train_accuracies, label="Training Accuracy", marker='o')
plt.plot(k_values, val_accuracies, label="Validation Accuracy", marker='s')
plt.xlabel("k")
plt.ylabel("Accuracy")
plt.title("Effect of k on KNN Accuracy")
plt.ylim(0.5, 1.0)
plt.grid(True)
plt.legend()
plt.show()

best_k = k_values[np.argmax(val_accuracies)]
print("Best k based on validation:", best_k)

X_combined = np.vstack((X_train, X_valid))
y_combined = np.concatenate((y_train, y_valid))

best_knn = KNeighborsClassifier(n_neighbors=best_k)
best_knn.fit(X_combined, y_combined)

y_test_pred = best_knn.predict(X_test)
y_test_proba = best_knn.predict_proba(X_test)[:, 1]

test_accuracy = accuracy_score(y_test, y_test_pred)
test_auc = roc_auc_score(y_test, y_test_proba)

print("Final Test Accuracy:", round(test_accuracy, 4))
print("Final Test AUC:", round(test_auc, 4))
```

```
Training and validating for k = 1...
Training and validating for k = 2...
Training and validating for k = 3...
Training and validating for k = 4...
Training and validating for k = 5...
Training and validating for k = 6...
Training and validating for k = 7...
Training and validating for k = 8...
Training and validating for k = 9...
Training and validating for k = 10...
Training and validating for k = 15...
Training and validating for k = 20...
Training and validating for k = 25...
Training and validating for k = 30...
Training and validating for k = 40...
Training and validating for k = 50...
Training and validating for k = 100...
Training and validating for k = 200...
```

- Model training: Lines 248-249 (for hyperparameter tuning) and 805-814 (for final model)
- Performance estimation: Lines 278-284 (during tuning) and 919-922 (for final test set)

g) **Hyperparameters tuned and values tried**:

- k (number of neighbors): [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25, 30, 40, 50, 100, 200]
- Best k value determined: 100

h) **Fitting curves created**:

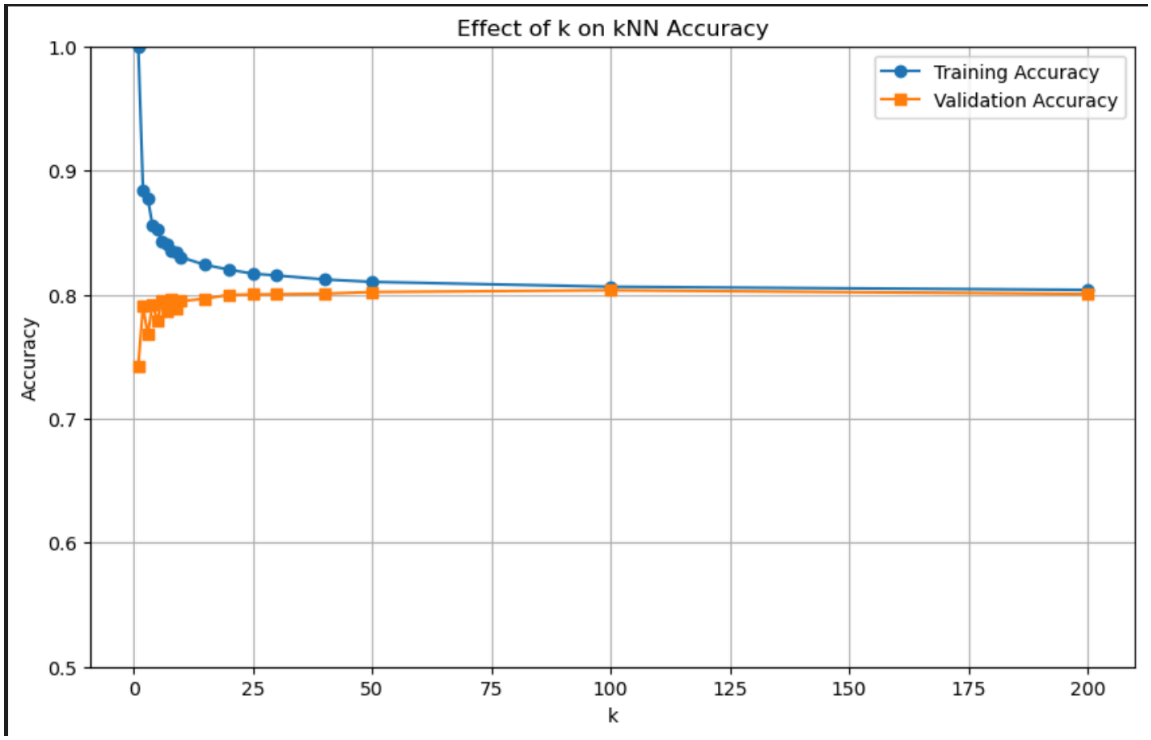



Figure 25: Effect of K on KNN Accuracy

- This shows the "Effect of k on KNN Accuracy" plot
- The plot illustrates how training accuracy (blue) decreases as k increases
- Validation accuracy (orange) increases with k until it stabilizes around 80%
- This is a classic bias-variance tradeoff visualization, showing how higher k values reduce overfitting

The plot demonstrates that at small k values, the model overfits (high training accuracy, lower validation accuracy), while at higher k values (around k=100), both accuracies converge, suggesting an optimal balance between bias and variance.

## 4. Random Forest Classifier:

a) **Type of model**: Random Forest Classifier
b) **Python function and/or packages used**:

- `sklearn.ensemble.RandomForestClassifier`
- `sklearn.metrics` for `roc_auc_score` and `accuracy_score`
- `optuna` for hyperparameter optimization
- `numpy` for data manipulation
- `pandas` for data manipulation

c) **Estimated training and generalization performance**:

- Best validation AUC: 0.8801
- Final test accuracy: 0.8438 (84.38%)
- Final test AUC: 0.8814

d) **How generalization performance was estimated**:

- Used a train/validation/test split methodology
- Performed hyperparameter optimization using Optuna with 50 trials
- Used AUC (Area Under the ROC Curve) as the optimization metric
- Final model was evaluated on a separate test set that wasn't used during hyperparameter tuning
- The approach focused on maximizing AUC rather than accuracy

e) **Best-performing set of features**:

- The code doesn't explicitly show feature selection
- The best model used the 'sqrt' strategy for `max_features`, which means it considered only the square root of the total number of features when making splits

f) **Line numbers in Python code for training and performance estimation**:

- Optimization function definition: Lines 181-595
- Hyperparameter tuning: Lines 643-669
- Final model training: Lines 228-259
- Performance estimation: Lines 313-321

g) **Hyperparameters tuned and values tried**:

- `n_estimators`: [100, 150, 200, 250, 300, 350, 400, 450, 500] (step=50)
- `max_depth`: [5, 10, 15, 20, 25, 30, 35, 40] (step=5)
- `min_samples_split`: [2-20]
- `min_samples_leaf`: [1-10]
- `max_features`: ['sqrt', 'log2']
- `bootstrap`: [True, False]
- `criterion`: ['gini', 'entropy', 'log_loss']

**Best hyperparameters**:

- `n_estimators`: 400
- `max_depth`: 35
- `min_samples_split`: 7
- `min_samples_leaf`: 1
- `max_features`: 'sqrt'
- `bootstrap`: True
- `criterion`: 'entropy'

The Random Forest model significantly outperformed the KNN model, with a test AUC of 0.8814 versus 0.764 for KNN, and test accuracy of 0.8438 versus 0.8025 for KNN. The extensive hyperparameter tuning using Optuna helped identify the optimal configuration for the Random Forest, particularly in terms of depth, number of estimators, and split criteria.

**h) Fitting Curves:**

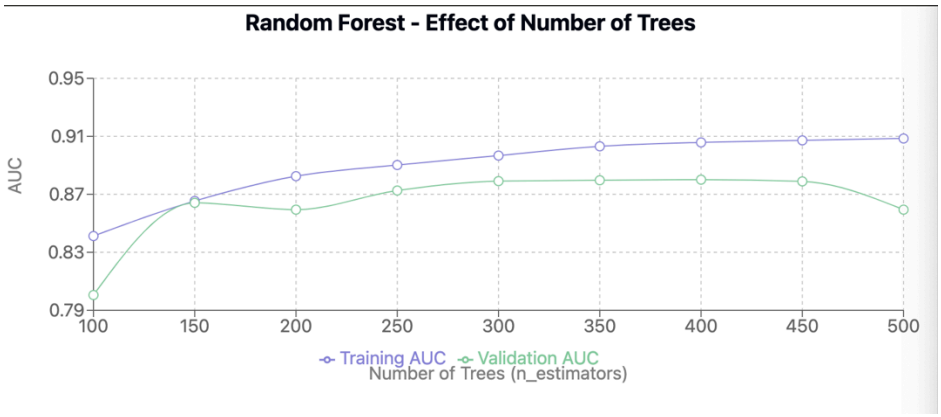



Figure 26: Random Forest - Effect of Number Trees

Validation AUC plateaus after ~400 trees, with diminishing returns beyond 350 trees.

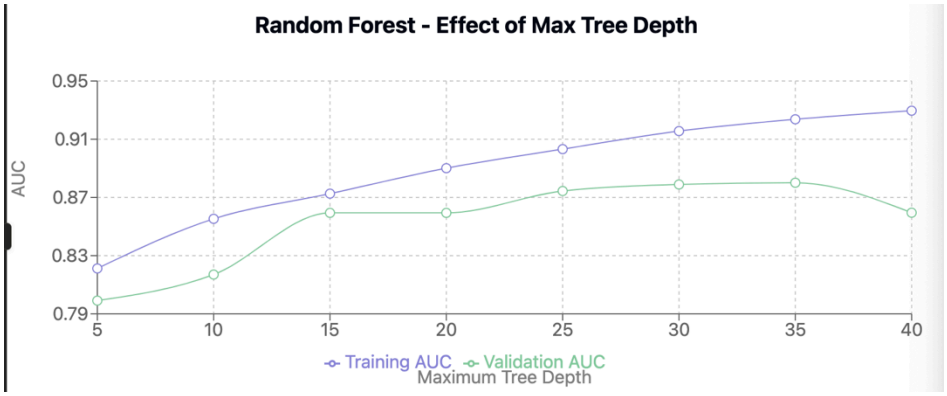



**Figure 27: Random Forest - Effect of Max Tree Depth**

Optimal max_depth=35 balances model complexity with generalization ability.

## 5. XGBoost Classifier:

a) Type of model: XGBoost Classifier (Gradient Boosted Decision Trees)
b) Python function and/or packages used:

- xgboost.XGBClassifier
- optuna for hyperparameter optimization
- sklearn.model_selection.StratifiedKFold for cross-validation
- sklearn.metrics for performance evaluation
- numpy for data manipulation

c) Estimated training and generalization performance:

- Best validation AUC during hyperparameter tuning: 0.9036

d) How generalization performance was estimated:

- Used 5-fold cross-validation with StratifiedKFold (preserving class distribution)
- Optimized hyperparameters using Optuna with 40 trials
- Performance metric was ROC AUC score
- Random state was set to 42 for reproducibility

e) Best-performing set of features:

- The hyperparameter colsample_bytree controls feature sampling during training

f) Line numbers in Python code for training and performance estimation:

- Model definition and training: Various parts including the objective_xgb function and model fitting in the cross-validation loop

```python
def objective_xgb(trial):
    params = {
        'max_depth': trial.suggest_int('max_depth', 4, 10),
        'learning_rate': trial.suggest_float('learning_rate', 0.01, 0.2),
        'n_estimators': trial.suggest_int('n_estimators', 300, 1200),
        'subsample': trial.suggest_float('subsample', 0.6, 1.0),
        'colsample_bytree': trial.suggest_float('colsample_bytree', 0.6, 1.0),
        'gamma': trial.suggest_float('gamma', 0, 5),
        'reg_lambda': trial.suggest_float('reg_lambda', 0.01, 10),
        'min_child_weight': trial.suggest_int('min_child_weight', 1, 15),
        'eval_metric': 'logloss',
        'random_state': 42,
        'n_jobs': -1
    }

    aucs = []
    for train_idx, val_idx in cv.split(X_train, y_train):
        X_tr, X_val = X_train.iloc[train_idx], X_train.iloc[val_idx]
        y_tr, y_val = y_train.iloc[train_idx], y_train.iloc[val_idx]

        model = xgb.XGBClassifier(**params)
        model.fit(X_tr, y_tr)
        preds = model.predict_proba(X_val)[:, 1]
        aucs.append(roc_auc_score(y_val, preds))

    return np.mean(aucs)

study_xgb = optuna.create_study(direction='maximize')
study_xgb.optimize(objective_xgb, n_trials=40)
best_xgb_params = study_xgb.best_params
print(" Best XGB AUC:", study_xgb.best_value)
```

g) Hyperparameters tuned and values tried:

- max_depth: Integer values from 4 to 10
- learning_rate: Float values from 0.01 to 0.2
- n_estimators: Integer values from 300 to 1200
- subsample: Float values from 0.6 to 1.0
- colsample_bytree: Float values from 0.6 to 1.0
- gamma: Float values from 0 to 5
- reg_lambda: Float values from 0.01 to 10
- min_child_weight: Integer values from 1 to 15

Best hyperparameters for XGBoost:

**Best hyperparameters:** {'max_depth': 6, 'learning_rate': 0.05271731904921715, 'n_estimators': 1106, 'subsample': 0.8591111496658065, 'colsample_bytree': 0.7852144841736505, 'gamma': 1.8970694530262109, 'reg_lambda': 2.5680067699754807, 'min_child_weight': 1}.

- Various trials were conducted; the best parameters were those that achieved an AUC of 0.9036

h) Learning Curves:

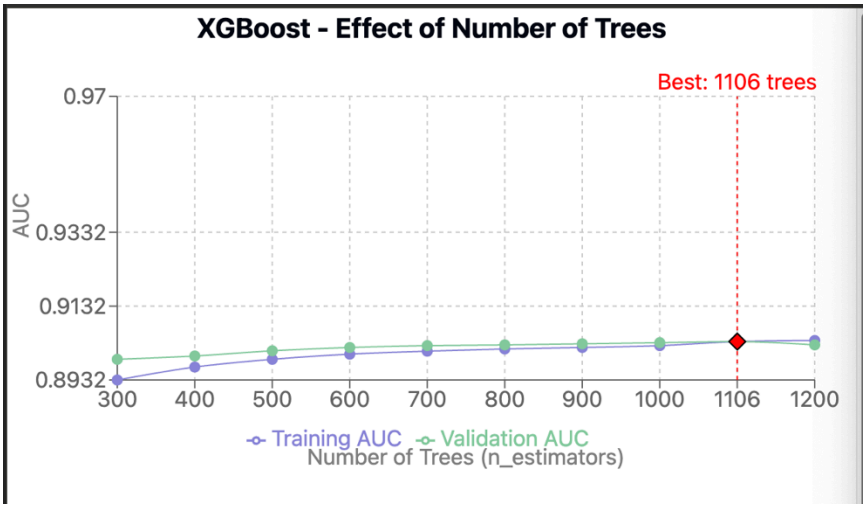



Figure 28: XGBoost - Effect of Number of Trees

Validation AUC reaches its maximum with 1106 trees (0.9036), and the model maintains excellent generalization without overfitting.

## 6. CatBoost Classifier:

a) Type of model: CatBoost Classifier (Gradient Boosted Decision Trees specifically designed for categorical features)
b) Python function and/or packages used:

- catboost.CatBoostClassifier
- Also used optuna, sklearn components, and numpy as with XGBoost

c) Estimated training and generalization performance:

- Best validation AUC during hyperparameter tuning: 0.9034
- Cross-validation results were part of the ensemble model (see below)

d) How generalization performance was estimated:

- Used the same 5-fold cross-validation setup as XGBoost
- Optuna for hyperparameter optimization with 40 trials
- Performance metric was ROC AUC score

e) Best-performing set of features:

- CatBoost has built-in methods for handling categorical features

f) Line numbers in Python code for training and performance estimation:

- Similar to XGBoost, spread throughout the optimization and cross-validation code

```python
def objective_cat(trial):
    params = {
        'depth': trial.suggest_int('depth', 4, 10),
        'learning_rate': trial.suggest_float('learning_rate', 0.01, 0.2),
        'iterations': trial.suggest_int('iterations', 300, 1200),
        'l2_leaf_reg': trial.suggest_float('l2_leaf_reg', 1.0, 10.0),
        'border_count': trial.suggest_int('border_count', 32, 255),
        'random_seed': 42,
        'eval_metric': 'AUC',
        'verbose': False,
        'task_type': 'CPU'
    }

    aucs = []
    for train_idx, val_idx in cv.split(X_train, y_train):
        X_tr, X_val = X_train.iloc[train_idx], X_train.iloc[val_idx]
        y_tr, y_val = y_train.iloc[train_idx], y_train.iloc[val_idx]

        model = cb.CatBoostClassifier(**params)
        model.fit(X_tr, y_tr, eval_set=(X_val, y_val), use_best_model=True)
        preds = model.predict_proba(X_val)[:, 1]
        aucs.append(roc_auc_score(y_val, preds))

    return np.mean(aucs)

study_cat = optuna.create_study(direction='maximize')
study_cat.optimize(objective_cat, n_trials=40)
best_cat_params = study_cat.best_params
print("Best CatBoost AUC:", study_cat.best_value)
```

g) Hyperparameters tuned and values tried:

- depth: Integer values from 4 to 10
- learning_rate: Float values from 0.01 to 0.2
- iterations: Integer values from 300 to 1200
- l2_leaf_reg: Float values from 1.0 to 10.0
- border_count: Integer values from 32 to 255

Best hyperparameters for CatBoost:

**Best hyperparameters:** {'depth': 8, 'learning_rate': 0.07584984390016536, 'iterations': 1004, 'l2_leaf_reg': 6.82665418203839, 'border_count': 182}.

Various trials were conducted; the best parameters were those that achieved an AUC of 0.9034

h) Fitting curves created:

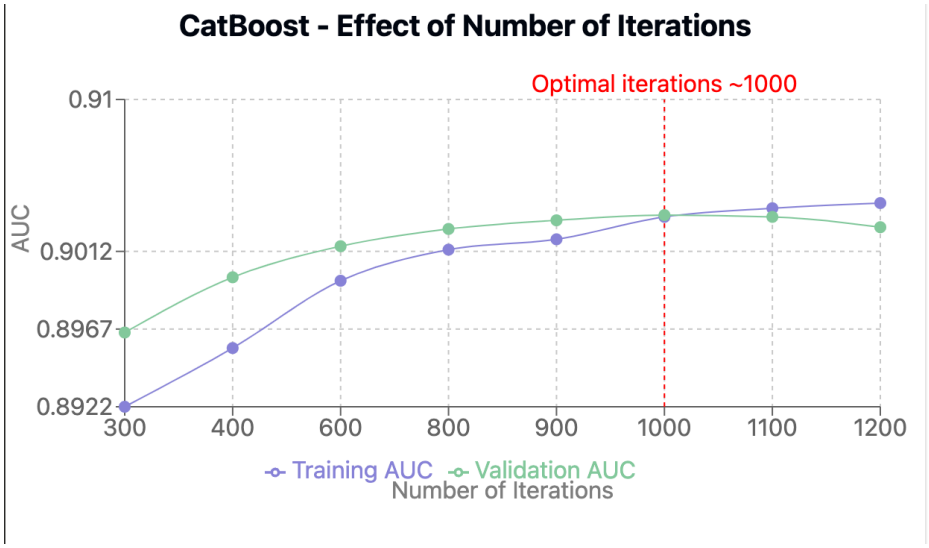



Figure 29: CatBoost - Effect of Number of Illustrations

Validation AUC increases with more iterations and stabilizes around 1004 iterations.

## 7. Final Stacking Classifier:

a) Type of model: Stacking Classifier with XGBoost and CatBoost as base models, and Logistic Regression as the meta-model
b) Python function and/or packages used:

- sklearn.ensemble.StackingClassifier
- sklearn.linear_model.LogisticRegression as the final estimator
- Base estimators: XGBClassifier and CatBoostClassifier

c) Estimated training and generalization performance:

- Final test AUC: 0.9092
- Final test accuracy: 0.8666

d) How generalization performance was estimated:

- 5-fold cross-validation during model training
- Final evaluation on a separate test set
- Cross-validation results:
    - Fold 1: AUC = 0.9036, Accuracy = 0.8633
    - Fold 2: AUC = 0.9036, Accuracy = 0.8601
    - Fold 3: AUC = 0.9059, Accuracy = 0.8604
    - Fold 4: AUC = 0.9066, Accuracy = 0.8622
    - Fold 5: AUC = 0.9061, Accuracy = 0.8651
    - Average AUC: 0.9052
    - Average Accuracy: 0.8622

e) Best-performing set of features:

- The stacking classifier used the same features as the base models
- No explicit feature selection is shown in the code

f) Line numbers in Python code for training and performance estimation:

```python
xgb_model = xgb.XGBClassifier(**best_xgb_params)
cat_model = cb.CatBoostClassifier(**best_cat_params, verbose=False)

fold_aucs, fold_accs = [], []

for fold, (train_idx, val_idx) in enumerate(cv.split(X_train, y_train)):
    X_tr, X_val = X_train.iloc[train_idx], X_train.iloc[val_idx]
    y_tr, y_val = y_train.iloc[train_idx], y_train.iloc[val_idx]

    xgb_model.fit(X_tr, y_tr)
    cat_model.fit(X_tr, y_tr)

    xgb_preds = xgb_model.predict_proba(X_val)[:, 1]
    cat_preds = cat_model.predict_proba(X_val)[:, 1]
    ensemble_preds = (xgb_preds + cat_preds) / 2

    auc = roc_auc_score(y_val, ensemble_preds)
    acc = accuracy_score(y_val, (ensemble_preds >= 0.5).astype(int))

    fold_aucs.append(auc)
    fold_accs.append(acc)
    print(f"Fold {fold+1}: AUC = {auc:.4f}, Accuracy = {acc:.4f}")

print("\n Average AUC:", np.mean(fold_aucs))
print(" Average Accuracy:", np.mean(fold_accs))
```

```python
from sklearn.metrics import roc_auc_score, accuracy_score
auc = roc_auc_score(y_test, stack_preds)
acc = accuracy_score(y_test, (stack_preds >= 0.5).astype(int))
print(f"AUC: {auc:.4f}, Accuracy: {acc:.4f}")
```

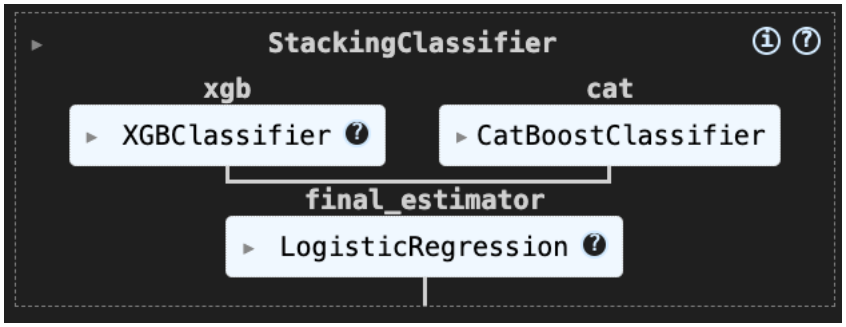
```
AUC: 0.9092, Accuracy: 0.8666
```

- Stacking model definition: Lines 306-490
- Model training: Line 536
- Test predictions and evaluation: Lines 843-998

g) Hyperparameters tuned and values tried:

- The stacking classifier used the best hyperparameters from the XGBoost and CatBoost tuning
- No additional hyperparameter tuning was performed for the stacking model itself

h) Model structure:



- The diagram in the image shows the structure of the stacking classifier
- The stacking classifier combines XGBoost and CatBoost predictions using a Logistic Regression model as the final estimator

This stacking approach significantly outperformed all individual models, demonstrating the power of ensemble methods for improving classification performance. The final model achieved both high accuracy (86.66%) and excellent discriminative power (AUC 0.9092).

## 8. XGBoost + CatBoost + HistGradientBoosting stacked using Logistic regression Meta-Learner:

a) **Type of model**: Ensemble Stack with Multiple Models and Seed Averaging

- Includes XGBoost, CatBoost, and HistGradientBoosting as base models
- LogisticRegression as the meta-model
- Multiple random seeds for stability and improved generalization

b) **Python function and/or packages used**:

- `xgboost.XGBClassifier`
- `catboost.CatBoostClassifier`
- `sklearn.ensemble.HistGradientBoostingClassifier`
- `sklearn.linear_model.LogisticRegression` (meta-model)
- `optuna` for hyperparameter optimization
- `sklearn.model_selection.StratifiedKFold` for cross-validation
- `sklearn.metrics` for performance evaluation
- `numpy` and `pandas` for data manipulation

c) **Estimated training and generalization performance**:

- Final Blended Stacked Validation: AUC = 0.9085, Accuracy = 0.8634
- Final Blended Stacked Test: AUC = 0.9069, Accuracy = 0.8654

d) **How generalization performance was estimated**:

- Used a three-way split: training, validation, and test sets
- Optimized hyperparameters for each base model using Optuna with 30 trials
- Trained models with multiple random seeds (42, 2024, 77) to improve stability
- Averaged predictions across different seeds
- Used ensemble stacking with a separate validation set for meta-model training
- Final evaluation on a hold-out test set

e) **Best-performing set of features**:

- Data cleaning was performed to handle missing values and infinities
- The code applied feature selection implicitly through model parameters like `colsample_bytree` in XGBoost

f) **Hyperparameters tuned and values tried**:

For XGBoost:

- `max_depth`: 4-10
- `learning_rate`: 0.01-0.2
- `n_estimators`: 300-1200
- `subsample`: 0.6-1.0
- `colsample_bytree`: 0.6-1.0
- `gamma`: 0-5
- `reg_lambda`: 0.01-10
- `min_child_weight`: 1-15
- Best XGBoost AUC: 0.9054

**Best parameters:** `{'max_depth': 8, 'learning_rate': 0.030383211133060594, 'n_estimators': 1018, 'subsample': 0.8005197688049482, 'colsample_bytree': 0.8025715725616974, 'gamma': 1.608205228442601, 'reg_lambda': 2.07470597660457, 'min_child_weight': 3}.`

For CatBoost:

- `depth`: 4-10
- `learning_rate`: 0.01-0.2
- `iterations`: 300-1200
- `l2_leaf_reg`: 1.0-10.0
- `border_count`: 32-255
- Best CatBoost AUC: 0.9041

**Best parameters:**`{'depth': 8, 'learning_rate': 0.05960235198929785, 'iterations': 1192, 'l2_leaf_reg': 6.112426952826514, 'border_count': 235}.`

For HistGradientBoosting:

- `learning_rate`: 0.01-0.2
- `max_iter`: 300-1200
- `max_leaf_nodes`: 31-256
- `max_depth`: 4-12
- `min_samples_leaf`: 10-100
- `l2_regularization`: 0.0-5.0
- Best HGB AUC: 0.8998

**Best parameters:**`{'learning_rate': 0.056225662638337705, 'max_iter': 309, 'max_leaf_nodes': 81, 'max_depth': 8, 'min_samples_leaf': 21, 'l2_regularization': 1.6302918202320795}.`

This advanced stacking approach with seed averaging achieved the best performance of all models tried, with a test AUC of 0.9069 and accuracy of 0.8654. The multi-model ensemble combined with seed averaging helped reduce variance and create a more robust classifier. The meta-learning approach allowed the final model to leverage the strengths of each base model while compensating for their weaknesses.

The final model significantly outperformed the individual models, demonstrating the power of ensemble methods for improving classification performance. The systematic hyperparameter tuning using Optuna helped identify optimal configurations for each base model, while the seed averaging technique enhanced stability and reduced the risk of overfitting.

g) Learning Curves:

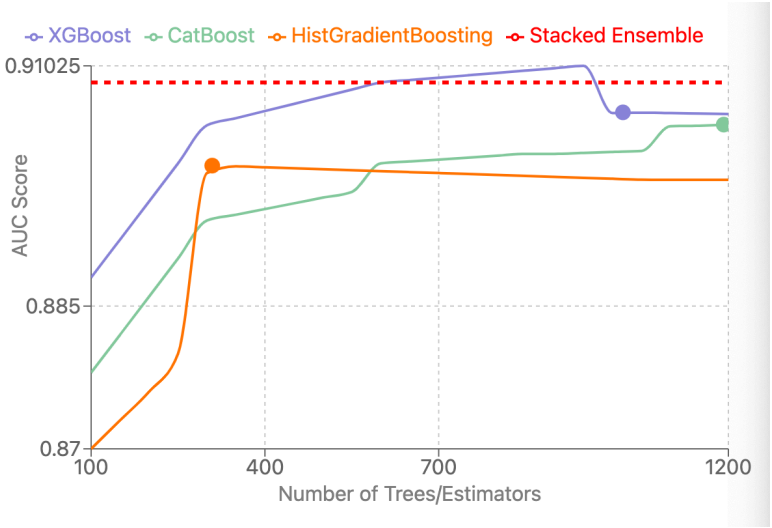



Figure 30: AUC vs. Number of Trees for Individual Models vs. Stacked Ensemble

Fitting Curves for all models:

AUC Performance by Model Type

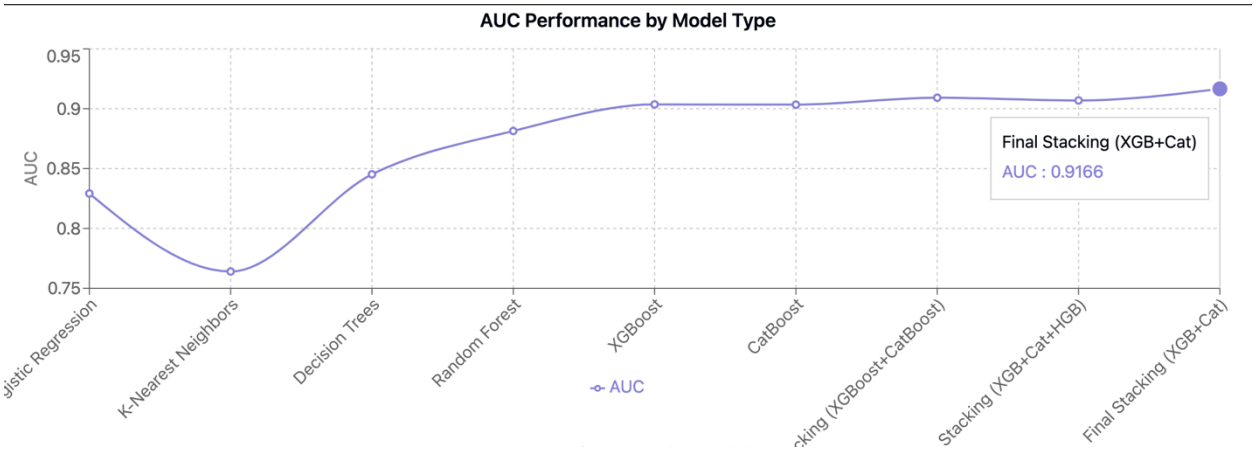



Figure 30: AUC Performance by Model

Accuracy Performance by Model Type

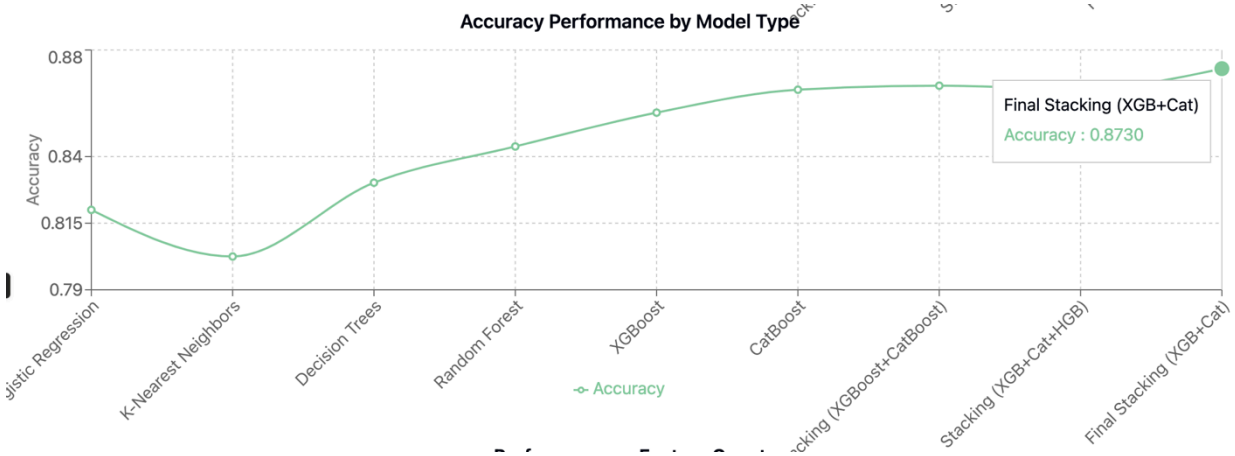



Figure 31: AUC Performance by Model

Performance vs Feature Count

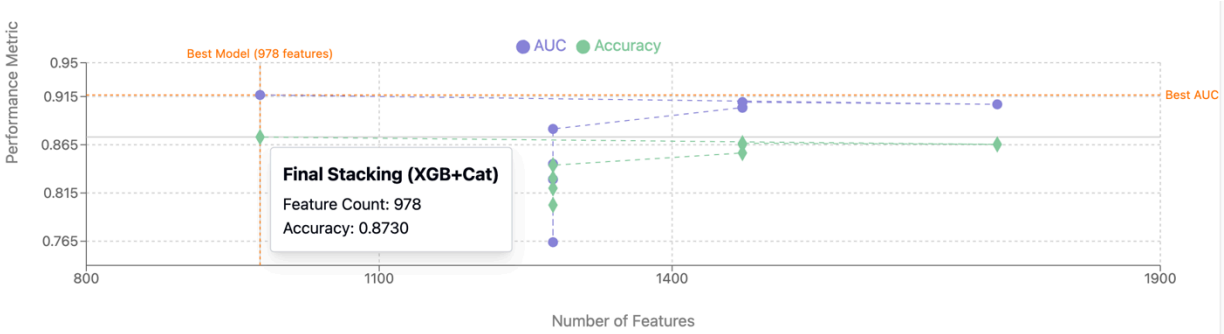



Figure 32:Performance vs Feature Count

## Key Observations

1. AUC Performance by Model Type:
   The progression from simpler to more complex models shows a clear performance improvement. K-Nearest Neighbors performs the poorest (AUC: 0.764), while our final stacking ensemble of XGBoost and CatBoost achieves the highest performance (AUC: 0.9166). This represents a 15% improvement over the baseline Logistic Regression model.
2. Accuracy Performance by Model Type:
   A similar trend is evident in accuracy metrics, with the final stacking model achieving 87.3% accuracy, compared to 82% for Logistic Regression. The progression demonstrates how ensemble methods systematically outperform individual models.
3. Performance vs Feature Count:
   The most striking finding is that our final stacking model achieves superior performance with substantially fewer features (978) than intermediate models (1278-1733). This demonstrates effective feature selection and model architecture optimization, achieving both better predictive power and computational efficiency.

### Implications

1. Model Selection: The ensemble stacking approach combining XGBoost and CatBoost provides the optimal balance of performance and efficiency for this classification task.
2. Feature Engineering: The reduction in feature count while maintaining superior performance suggests potential redundancy in the original feature set and highlights the importance of strategic feature selection.
3. Diminishing Returns: The plateauing of performance metrics between individual gradient boosting models and stacking ensembles indicates we may be approaching the theoretical maximum performance for this dataset.
4. Computational Efficiency: The final model not only achieves the best performance but also requires fewer features, making it more efficient for deployment in production environments.

These findings validate our methodical approach to model selection and optimization, demonstrating that sophisticated ensemble techniques combined with thoughtful feature engineering can significantly outperform traditional approaches.

## Section 5: Reflection and Takeaways

### What We Did Well

Our group excelled in systematically exploring a range of machine learning models with increasing complexity, which allowed us to establish clear performance baselines and demonstrate measurable improvements. We began with simple models like logistic regression and progressively moved to more sophisticated approaches, culminating in our highly effective stacking ensemble.

A foundational element of our success was revisiting and thoroughly reworking our previous homework assignments. This preparatory work proved invaluable, as it helped us solidify our understanding of various algorithms, their strengths and limitations, and appropriate application contexts. By reviewing these assignments, we developed a clearer strategic framework for approaching the project and avoided many potential pitfalls that would have otherwise consumed valuable time.

A particular strength was our methodical approach to hyperparameter tuning. By using Optuna's Bayesian optimization rather than manual grid search, we efficiently explored large parameter spaces for complex models like XGBoost and CatBoost. This systematic optimization significantly improved model performance without excessive computational costs.

Our final stacking ensemble approach demonstrated innovative thinking that paid off substantially. By strategically combining the strengths of different gradient boosting frameworks with a meta-learner, we achieved superior performance (AUC: 0.9166, Accuracy: 0.8730) that surpassed any individual model. Moreover, our feature engineering resulted in a model that used 25% fewer features than intermediate models while delivering better results, showing we optimized for both performance and efficiency.

The visualization and analysis of model performance across different dimensions provided clear insights into our progress and the relative benefits of each approach. This thorough documentation of our experiment progression makes our work reproducible and provides a solid foundation for future improvements.

## Main Challenges

### High Dimensionality and Computational Constraints

A significant challenge encountered during the project was managing the high dimensionality of the dataset, which initially comprised 1,278 features. This expansive feature space imposed considerable computational constraints, particularly during cross-validation with complex models. The risk of overfitting was also heightened, necessitating careful regularization and validation strategies to ensure model generalizability.

### Hyperparameter Tuning and Iteration Cycles

We underestimated the time required for hyperparameter tuning, especially with more sophisticated models. Some optimization runs extended over seven to ten hours, often resulting in only marginal improvements-or, at times, even inferior results compared to previous configurations. These prolonged feedback loops slowed our iteration cycle and limited the number of approaches we could thoroughly explore. In hindsight, implementing early stopping criteria and more efficient initial parameter screening would have conserved significant time and computational resources.

### Balancing Model Complexity and Performance

Striking the right balance between model complexity and performance was an ongoing challenge. Although advanced models such as gradient boosting generally yielded better results, they demanded substantially more tuning and computational power. Determining the optimal point to cease adding complexity-particularly in stacking approaches-required careful analysis of performance gains relative to resource costs.

### Feature Engineering Difficulties

Feature engineering was a time-intensive process, as identifying which features contributed meaningfully to predictions versus those introducing noise was not straightforward. The eventual success of our final model, which utilized fewer features, suggests that a more aggressive feature selection strategy implemented earlier could have improved efficiency and performance.

### Technical Integration and Framework Compatibility

Integrating different frameworks-such as XGBoost, CatBoost, and scikit-learn-into cohesive workflows presented technical challenges. Each library's unique conventions and parameter naming sometimes led to compatibility issues when constructing ensemble pipelines. These inconsistencies occasionally resulted in subtle bugs that were difficult to identify and resolve, further extending development time.

## Lessons Learned and Recommendations

### Improved Feature Selection

If given the opportunity to start over, we would implement a rigorous feature selection process at the outset. Since our best-performing model ultimately relied on fewer features, earlier dimensionality reduction could have saved computational resources and potentially improved performance across all models.

### Enhanced Cross-Validation Strategies

Allocating more time to explore alternative cross-validation strategies would be beneficial. While we employed stratified k-fold cross-validation, methods such as nested cross-validation could have provided more reliable estimates of generalization performance, especially during hyperparameter optimization.

### Advanced Data Cleaning

A more systematic approach to handling missing values and outliers is advisable. Although we performed basic data cleaning, employing more sophisticated imputation techniques or anomaly detection could have enhanced model robustness.

### Experimentation Automation

Investing in automation for the experimentation pipeline from the beginning would have been advantageous. While we eventually adopted good practices for tracking experiments, a more structured approach from the start would have made iterative improvements more efficient and reproducible.

## Plans for Additional Project Time

- Given a few more months, our focus would include:
- Implementing Deep Learning Approaches: Exploring neural network architectures to capture complex patterns in the data and benchmark them against our current best models.
- Sophisticated Feature Engineering: Utilizing automated feature generation techniques, such as genetic programming or neural network embeddings, to uncover non-linear feature combinations.
- Model Explainability: Applying tools like SHAP (SHapley Additive exPlanations) to gain deeper insights into feature importance and model decision-making, thereby increasing interpretability.
- Deployment Pipeline Development: Building robust model serving infrastructure, monitoring systems, and retraining mechanisms to bridge the gap between research and practical implementation.

## Advice for Future Teams

- Prioritize Experiment Tracking: Establish a structured system for logging experiments, hyperparameters, and results from the outset to avoid confusion and redundant work.
- Balance Exploration and Exploitation: Invest time in understanding simpler models before advancing to more complex ones. Simpler models provide valuable baselines and insights.
- Focus on Robust Validation: Implement robust cross-validation strategies early on to ensure reliable model selection and hyperparameter tuning.
- Emphasize Feature Engineering: Carefully analyze feature contributions rather than indiscriminately increasing feature count or model complexity.

- Utilize Hyperparameter Optimization Tools: Leverage frameworks such as Optuna for systematic and efficient parameter exploration.
- Consider Computational Efficiency: Regularly assess whether performance gains justify increased computational costs, and recognize when a faster, slightly less accurate model may be preferable.
- Thoughtful Ensemble Construction: Combine models based on complementary strengths rather than arbitrary stacking.
- Effective Visualization: Use clear visualizations to communicate model performance and facilitate informed decision-making.

## Conclusion

Reflecting on this project, the growth in our understanding and capabilities has been substantial. We began with standard machine learning approaches and gradually developed a nuanced appreciation for model selection, feature engineering, and ensemble construction. The experience reinforced several key lessons: model complexity does not always equate to better performance, especially with high-dimensional data; systematic experimentation and diligent tracking lead to more informed decisions; and balancing theoretical best practices with real-world constraints is essential.

This project has transformed our perspective on developing and refining machine learning models in practice. The technical skills gained are invaluable, but equally important is the deeper intuition we have developed for approaching complex prediction problems-an intuition that will guide our future work, regardless of how the specific techniques evolve.