XGBoost Regression Model Report: Airbnb Monthly Revenue Prediction

Model Overview

- **Algorithm**: XGBoost (Extreme Gradient Boosting)
- **Objective**: Predict monthly revenue for Airbnb listings
- **Model Type**: Regression

Feature Engineering Techniques

1. **Name Feature Extraction**
   - Extracted features from listing names
   - Parsed information about:
     - Number of bedrooms
     - Number of bathrooms
     - Private/shared bath
     - Overall rating
     - New property status
2. **Neighborhood Overview Processing**
   - Cleaned text descriptions
   - Applied TF-IDF vectorization
   - Extracted 100 most important text features
3. **Advanced Feature Engineering**
   - Created revenue-related features
     - Minimum monthly revenue
     - Maximum monthly revenue
     - Revenue per booking
   - Computed availability ratios
   - Calculated distance from city center
   - Generated monthly revenue projections

Preprocessing Pipeline

- **Numerical Features**
  - Median imputation
  - Standard scaling

    - Includes engineered features and TF-IDF vectors

- **Categorical Features**
    - Constant value imputation
    - One-hot encoding
    - Handles unknown categories

Model Hyperparameters

```
XGBRegressor(
    random_state=42,
    colsample_bytree=0.6661,
    learning_rate=0.0147,
    max_depth=3,
    min_child_weight=1,
    n_estimators=235,
    subsample=0.6022
)
```

Performance Metrics

Training Performance

- **Mean Squared Error (MSE)**: 1,175,188.32
- **R-squared (R2)**: 0.3541

Validation Performance

- **Mean Squared Error (MSE)**: 1,180,195.36
- **R-squared (R2)**: 0.2732

Model Interpretation

- Moderate predictive power with R2 around 0.35
- Indicates significant variability in monthly revenue
- Captures about 35% of the variance in the training data
- Slight overfitting (small difference between train and validation R2)

Potential Improvements

1. Feature engineering
   - Create more interaction features
   - Explore non-linear transformations
2. Hyperparameter tuning
   - More extensive grid/random search
   - Consider ensemble methods
3. Advanced techniques
   - Feature selection
   - Try other algorithms (Random Forest, Gradient Boosting)

Key Takeaways

- XGBoost provides a solid baseline for predicting Airbnb monthly revenue
- Complex feature engineering significantly contributes to model performance
- Room for improvement through advanced modeling techniques

Source Code

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split, RandomizedSearchCV
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.linear_model import ElasticNet
import xgboost as xgb
from sklearn.metrics import mean_squared_error, r2_score
import re
from bs4 import BeautifulSoup
from sklearn.feature_extraction.text import TfidfVectorizer
from scipy.stats import randint, uniform
```

```python
import warnings
from haversine import haversine
warnings.filterwarnings('ignore')

def load_and_preprocess_data(train_path, test_path):
    # Load data
    train_df = pd.read_csv(train_path)
    test_df = pd.read_csv(test_path)

    # Drop unnecessary columns and set index
    if 'Unnamed: 0' in train_df.columns:
        train_df.drop('Unnamed: 0', axis=1, inplace=True)
    if 'Unnamed: 0' in test_df.columns:
        test_df.drop('Unnamed: 0', axis=1, inplace=True)

    train_df.set_index('id', inplace=True)
    test_df.set_index('id', inplace=True)

    for df in [train_df, test_df]:
        for col in ['host_response_rate', 'host_acceptance_rate']:
            if col in df.columns:
                df[col] = df[col].str.replace('%', '').astype(float)

    return train_df, test_df

def extract_features_from_name(df):
    # Extract features from name column
    def extract_rating(parts):
        for part in parts:
            if ' ' in part:
                try:
                    return float(part.replace(' ', '').strip())
                except ValueError:
                    continue
        return 0

    def is_property_new(parts):
        for part in parts:
            if 'new' in part.lower():
                return 1
        return 0
```

```python
def extract_bedrooms(parts):
    for part in parts:
        if 'Studio' in part:
            return 0
        elif 'bedroom' in part:
            try:
                return int(part.split()[0])
            except ValueError:
                continue
    return 0

def extract_beds(parts):
    for part in parts:
        if 'bed' in part:
            try:
                return int(part.split()[0])
            except ValueError:
                continue
    return 0

def extract_baths(parts):
    for part in parts:
        if 'half-bath' in part.lower():
            return 0.5
        if 'bath' in part.lower():
            try:
                return float(part.split()[0])
            except ValueError:
                continue
    return 0

def is_private_bath(parts):
    for part in parts:
        if 'private' in part.lower() and 'bath' in part.lower():
            return 1
    return 0

def is_shared_bath(parts):
    for part in parts:
        if 'shared' in part.lower() and 'bath' in part.lower():
            return 1
    return 0
```

```python
    df["split_parts"] = df["name"].str.split("·")
    df["bedrooms"] = df["split_parts"].apply(extract_bedrooms)
    df["beds"] = df["split_parts"].apply(extract_beds)
    df["baths"] = df["split_parts"].apply(extract_baths)
    df["is_bath_private"] = df["split_parts"].apply(is_private_bath).astype(int)
    df["is_bath_shared"] = df["split_parts"].apply(is_shared_bath).astype(int)
    df["overall_rating"] = df["split_parts"].apply(extract_rating)
    df["is_new_property"] = df["split_parts"].apply(is_property_new).astype(int)

    df.drop('split_parts', axis=1, inplace=True)
    return df

def process_neighborhood_overview(df, tfidf_vectorizer=None):
    def clean_text(text):
        if pd.isna(text):
            return ''
        text = BeautifulSoup(text, "html.parser").get_text()
        text = re.sub(r'[^a-zA-Z\s]', '', text)
        text = text.lower()
        text = re.sub(r'\s+', ' ', text).strip()
        return text

    df['cleaned_neighborhood_overview'] = df['neighborhood_overview'].apply(
                                                            clean_text)
    df['cleaned_neighborhood_overview'].fillna('no description available',
                                               inplace=True)

    if tfidf_vectorizer is None:
        tfidf_vectorizer = TfidfVectorizer(max_features=100,
                                           stop_words='english',
                                           ngram_range=(1,5))
        tfidf_matrix = tfidf_vectorizer.fit_transform(df['cleaned_neighborhood_overview'])
    else:
        tfidf_matrix = tfidf_vectorizer.transform(df['cleaned_neighborhood_overview'])

    # Convert to DataFrame
    tfidf_df = pd.DataFrame(tfidf_matrix.toarray(),
                            columns=[f'tfidf_{i}' for i in range(100)],
                            index=df.index)

    return pd.concat([df, tfidf_df], axis=1), tfidf_vectorizer
```

```
def create_feature_engineering(df):
    # Clean price column
    df['price'] = df['price'].str.replace('$', '').str.replace(',', '').astype(float)

    df['instant_bookable'] = df['instant_bookable'].replace({'f': 0, 't': 1}).astype(int)

    # Create revenue features
    df['minimum_monthly_revenue'] = df['minimum_nights'] * df['price']
    df['maximum_monthly_revenue'] = df.apply(
        lambda row: (row['maximum_nights'] / 30) * row['price']
        if row['maximum_nights'] > 30
        else row['maximum_nights'] * row['price'],
        axis=1
    )
    df['revenue_per_booking'] = df['accommodates'] * df['price']

    # Create availability ratios
    availability_days = [30, 60, 90, 365]
    for days in availability_days:
        # Compute availability ratio
        df[f'availability_ratio_{days}'] = df[f'availability_{days}'] / days
        # Compute monthly revenue
        factor = 30 if days != 30 else 1
        df[f'monthly_revenue_{days}'] = df[f'availability_ratio_{days}'] * \
            df['price'] * factor

    # Define a central point (e.g., city center)
    central_point = (49.2789, -123.1195)  # Example: Downtown, Vancouver, BC

    # Calculate distance for each row
    df['distance_from_center'] = df.apply(lambda row: haversine((row['latitude'],
                                                                  row['longitude']),
                                                                 central_point),
                                                                 axis=1)



    return df

def prepare_features(df):
    # Columns to drop
    cols_to_drop = ['host_id', 'host_name', 'neighbourhood', 'longitude',
                    'latitude', 'amenities', 'name',
```

```
                'neighborhood_overview',
                'cleaned_neighborhood_overview']

    # Categorical columns for one-hot encoding
    categorical_features = ['host_response_time', 'neighbourhood_cleansed',
                          'property_type', 'room_type', 'host_is_superhost',
                          'instant_bookable']

    # Numerical columns for scaling
    numerical_features = ['host_response_rate', 'host_acceptance_rate',
                        'host_listings_count', 'host_total_listings_count',
                        'accommodates', 'beds', 'minimum_nights',
                        'maximum_nights', 'distance_from_center',
                        'minimum_nights_avg_ntm', 'maximum_nights_avg_ntm',
                        'number_of_reviews', 'number_of_reviews_ltm',
                        'review_scores_rating', 'review_scores_accuracy',
                        'review_scores_cleanliness', 'review_scores_checkin',
                        'review_scores_communication',
                        'review_scores_location',
                        'review_scores_value',
                        'calculated_host_listings_count',
                        'reviews_per_month', 'minimum_monthly_revenue',
                        'maximum_monthly_revenue', 'revenue_per_booking']

    # Add availability ratios and monthly revenue columns
    for days in [30, 60, 90, 365]:
        numerical_features.extend([f'availability_ratio_{days}',
                                 f'monthly_revenue_{days}'])

    # Add engineered features from name
    numerical_features.extend(['bedrooms', 'baths', 'overall_rating'])
    categorical_features.extend(['is_bath_private', 'is_bath_shared',
                               'is_new_property'])

    # Add TF-IDF features
    tfidf_features = [col for col in df.columns if col.startswith('tfidf_')]
    numerical_features.extend(tfidf_features)

    # Drop specified columns
    df = df.drop(cols_to_drop, axis=1)

    return df, numerical_features, categorical_features
```

```python
def create_model_pipeline(numerical_features, categorical_features):
    # Create preprocessors
    numeric_transformer = Pipeline(steps=[
        ('imputer', SimpleImputer(strategy='median')),
        ('scaler', StandardScaler())
    ])

    categorical_transformer = Pipeline(steps=[
        ('imputer', SimpleImputer(strategy='constant', fill_value='missing')),
        ('onehot', OneHotEncoder(handle_unknown='ignore', sparse_output=False))
    ])

    # Combine preprocessors
    preprocessor = ColumnTransformer(
        transformers=[
            ('num', numeric_transformer, numerical_features),
            ('cat', categorical_transformer, categorical_features)
        ])

    # Define parameter distributions for each model
    param_distributions = {
        'RandomForest': {
            'regressor__n_estimators': randint(100, 500),
            'regressor__max_depth': [None] + list(range(10, 50, 5)),
            'regressor__min_samples_split': randint(2, 20),
            'regressor__min_samples_leaf': randint(1, 10),
            'regressor__max_features': ['sqrt', 'log2', None]
        },
        'GradientBoosting': {
            'regressor__n_estimators': randint(100, 500),
            'regressor__learning_rate': uniform(0.01, 0.3),
            'regressor__max_depth': randint(3, 10),
            'regressor__min_samples_split': randint(2, 20),
            'regressor__min_samples_leaf': randint(1, 10),
            'regressor__subsample': uniform(0.6, 0.4)
        },
        'XGBoost': {
            'regressor__n_estimators': randint(100, 500),
            'regressor__learning_rate': uniform(0.01, 0.3),
            'regressor__max_depth': randint(3, 10),
            'regressor__min_child_weight': randint(1, 7),
            'regressor__subsample': uniform(0.6, 0.4),
```

```
        'regressor__colsample_bytree': uniform(0.6, 0.4)
    },
    'ElasticNet': {
        'regressor__alpha': uniform(0.0001, 1.0),
        'regressor__l1_ratio': uniform(0, 1),
        'regressor__max_iter': [2000]
    }
}

# Create base model pipelines
base_models = {
    'RandomForest': Pipeline([
        ('preprocessor', preprocessor),
        ('regressor', RandomForestRegressor(random_state=42))
    ]),
    'GradientBoosting': Pipeline([
        ('preprocessor', preprocessor),
        ('regressor', GradientBoostingRegressor(random_state=42))
    ]),
    'XGBoost': Pipeline([
        ('preprocessor', preprocessor),
        ('regressor', xgb.XGBRegressor(random_state=42,
                                       colsample_bytree=0.6661067756252009,
                                       learning_rate=0.01469092202235818,
                                       max_depth=3,
                                       min_child_weight=1,
                                       n_estimators=235,
                                       subsample=0.602208846849441))
    ]),
    'ElasticNet': Pipeline([
        ('preprocessor', preprocessor),
        ('regressor', ElasticNet(random_state=42))
    ])
}

# Create RandomizedSearchCV for each model
models = {}
models['XGBoost'] = base_models['XGBoost']
# for name, pipeline in base_models.items():
#     models[name] = RandomizedSearchCV(
#         pipeline,
#         param_distributions=param_distributions[name],
```

```
    #         n_iter=20,  # Number of parameter settings sampled
    #         cv=3,       # Number of cross-validation folds
    #         scoring='neg_root_mean_squared_error',
    #         n_jobs=-1,  # Use all available cores
    #         random_state=42,
    #         verbose=1
    #     )

    return models

def evaluate_models_and_create_submissions(models, X_train, y_train, X_test, test_df):
    results = {}
    best_params = {}

    for name, model in models.items():
        print(f"\nTraining {name} with RandomizedSearchCV...")

        # Fit model with randomized search on the full training data
        model.fit(X_train, y_train)

        # Store best parameters
        # best_params[name] = model.best_params_
        # print(f"\nBest parameters for {name}:")
        # for param, value in model.best_params_.items():
        #     print(f"{param}: {value}")

        # Evaluate model on the training data
        train_pred = model.predict(X_train)

        # Calculate metrics
        train_rmse = np.sqrt(mean_squared_error(y_train, train_pred))
        train_mse = mean_squared_error(y_train, train_pred)
        train_r2 = r2_score(y_train, train_pred)

        # Store results
        results[name] = {
            'Train RMSE': train_rmse,
            'Train MSE': train_mse,
            'Train R2': train_r2
            # 'Best CV Score': -model.best_score_  # Convert back from negative RMSE
        }
```

```python
        # Create submission file
        print(f"Creating submission for {name}...")

        # Retrain best model on full training data (optional if
        # model.best_estimator_ is already trained)
        # model.best_estimator_.fit(X_train, y_train)

        # Predict on test data
        # test_predictions = model.best_estimator_.predict(X_test)
        test_predictions = model.predict(X_test)

        # Create submission DataFrame
        submission = pd.DataFrame({
            'id': test_df.index,
            'monthly_revenue': test_predictions
        })

        # Save submission
        submission.to_csv(f'submission_{name.lower()}_tuned.csv', index=False)
        print(f"Saved submission_{name.lower()}_tuned.csv")

    return results, best_params

def create_validation_pipeline(X, y, models, test_size=0.2):
    """
    Creates a validation pipeline with train-validation split
    """
    # Create train-validation split
    X_train, X_val, y_train, y_val = train_test_split(
        X, y, test_size=test_size, random_state=42
    )

    validation_results = {}

    for name, model in models.items():
        print(f"\nTraining and validating {name}...")

        # Fit model on training data
        model.fit(X_train, y_train)

        # Make predictions on validation set
        val_pred = model.predict(X_val)
```

```
        train_pred = model.predict(X_train)

        # Calculate metrics
        train_rmse = np.sqrt(mean_squared_error(y_train, train_pred))
        train_mse = (mean_squared_error(y_train, train_pred))
        train_r2 = r2_score(y_train, train_pred)
        val_rmse = np.sqrt(mean_squared_error(y_val, val_pred))
        val_mse = (mean_squared_error(y_val, val_pred))
        val_r2 = r2_score(y_val, val_pred)

        # Store results
        validation_results[name] = {
            'Train MSE': train_mse,
            'Train R2': train_r2,
            'Validation MSE': val_mse,
            'Validation R2': val_r2
        }

    return validation_results, X_train, X_val, y_train, y_val

def main(selected_model=None):
    # Load data
    train_df, test_df = load_and_preprocess_data('input/train.csv', 'input/test.csv')

    # Process name column
    print("Processing name column...")
    train_df = extract_features_from_name(train_df)
    test_df = extract_features_from_name(test_df)

    # Process neighborhood overview
    print("Processing neighborhood overview...")
    train_df, tfidf_vectorizer = process_neighborhood_overview(train_df)
    test_df, _ = process_neighborhood_overview(test_df, tfidf_vectorizer)

    # Create engineered features
    print("Creating engineered features...")
    train_df = create_feature_engineering(train_df)
    test_df = create_feature_engineering(test_df)

    # Prepare features
    print("Preparing features...")
    train_df, numerical_features, categorical_features = prepare_features(train_df)
```

```python
    test_df, _, _ = prepare_features(test_df)

    # Prepare features
    X_train = train_df.drop('monthly_revenue', axis=1)
    y_train = train_df['monthly_revenue']
    X_test = test_df

    # Create model pipelines
    models = create_model_pipeline(numerical_features, categorical_features)

    # Filter for selected model
    if selected_model:
        if isinstance(selected_model, list):
            models = {name: model for name, model in models.items()
            if name in selected_model}
        else:
            models = {selected_model: models[selected_model]}

    # Perform validation
    validation_results, X_train_split, X_val, y_train_split, y_val = \
      create_validation_pipeline(
        X_train, y_train, models
    )

    # Print validation results
    print("\nValidation Results:")
    for model_name, metrics in validation_results.items():
        print(f"\n{model_name}:")
        for metric_name, value in metrics.items():
            print(f"{metric_name}: {value:.4f}")

    # Create final predictions and submission files
    results, best_params = evaluate_models_and_create_submissions(
        models, X_train, y_train, X_test, test_df
    )

    return validation_results, results

if __name__ == "__main__":
    main("XGBoost")
```