Project Title	Customer Satisfaction Prediction
Tools	Jupyter Notebook and VS code
Technologies	Machine learning , Python, Excel
Domain	Data Science
Project Difficulty level	Advanced

PHASE 01: Data Importing and Loading

Objective: In this phase, all required dependencies were successfully imported, and the raw dataset was loaded into a DataFrame for analysis. This forms the initial setup stage of the customer satisfaction prediction pipeline, ensuring that the environment is ready for data cleaning, feature engineering, and model development in the following phases.

1. **Library Imports:**

- Imported essential data manipulation and analysis libraries such as:
  - `pandas` and `numpy` for structured data handling and numerical operations.
  - `matplotlib.pyplot` and `seaborn` for data visualization.
  - `wordcloud` for generating visual word frequency representations.

- Included text processing and sentiment analysis tools:
  - `re` for regular expressions to clean text data.
  - `TextBlob` for extracting sentiment polarity from textual features.

- Imported machine learning utilities:
  - `scikit-learn` modules such as `train_test_split`, `GridSearchCV`, `TfidfVectorizer`, `OneHotEncoder`, `StandardScaler`, and `Pipeline` for feature engineering and model building.
  - `XGBClassifier` from `xgboost` for implementing an optimized gradient boosting model.

- Evaluation metrics including `accuracy_score`, `classification_report`, `roc_auc_score`, and confusion matrix visualization tools.

- Integrated SHAP (SHapley Additive exPlanations) for interpreting feature importance and model explainability.

- Suppressed non-critical warnings using `warnings.filterwarnings('ignore')` to maintain cleaner output during execution.

2. **Dataset Loading:**

- Defined the dataset path variable `DATA_PATH` pointing to the CSV file: `"customer_support_tickets.csv"`.

- Loaded the dataset using the `pandas.read_csv()` function.

- Verified successful loading by printing:
    - The shape of the dataset, showing the total number of records and columns.
    - The list of all column names for a quick overview of available features.

3. **Output Observation:**

- The dataset was successfully loaded with 8,469 records and 17 columns.
- The columns included a mix of customer details, ticket information, timestamps, textual fields, and the target variable:

```
['Ticket ID', 'Customer Name', 'Customer Email', 'Customer Age', 'Customer Gender',

 'Product Purchased', 'Date of Purchase', 'Ticket Type', 'Ticket Subject',

 'Ticket Description', 'Ticket Status', 'Resolution', 'Ticket Priority',

 'Ticket Channel', 'First Response Time', 'Time to Resolution',

 'Customer Satisfaction Rating']
```

PHASE 02: Data Cleaning and Feature Engineering

The main goal was to remove unnecessary columns, handle missing values, extract temporal features, engineer text-based variables, and define the target variable for classification.

1. Copying and Preparing the Data : This ensures that any transformations do not modify the source dataset, maintaining data integrity.

2. Dropping Irrelevant Identifiers: Columns such as `'Ticket ID'`, `'Customer Name'`, and `'Customer Email'` were removed because they contain unique or personally identifiable information that does not contribute to predicting customer satisfaction.

3. Combining Textual Information: A new column called `Ticket_Text` was created by concatenating three existing text columns — `'Ticket Subject'`, `'Ticket Description'`, and `'Resolution'`.
This step merges all relevant textual data related to customer complaints or inquiries into a single field for natural language processing (NLP).

4. Text Cleaning and Normalization:
The combined text was cleaned using a custom function `clean_text()`, which performs several operations:

- Converts text to lowercase.
- Removes URLs, punctuation, and non-alphanumeric characters.
- Replaces multiple spaces with a single space.

5. Extracting Date and Time-Based Features:
A function named `parse_time_features()` was used to extract valuable temporal features from columns like `'Date of Purchase'`, `'First Response Time'`, and `'Time to Resolution'`.
Key operations include:

- Converting dates to proper datetime format
- Deriving purchase year and purchase month
- Parsing response and resolution times
- Calculating `resolution_duration_hours` — the time difference between first response and resolution, expressed in hours
- Extracting response hour and day of week

These engineered time features help the model capture service responsiveness, which may strongly influence customer satisfaction.

6. Sentiment Analysis on Text:
TextBlob was applied on the `Ticket_Text` column to compute the sentiment polarity score for each ticket.
The polarity value ranges from `-1` (negative sentiment) to `+1` (positive sentiment), helping quantify how positively or negatively a customer expressed themselves.

This numeric sentiment feature can indicate dissatisfaction or approval tendencies within the ticket text.

7. Handling and Cleaning Numeric Columns:
The `'Customer Age'` column was converted to numeric format using `pd.to_numeric()`. Any invalid or missing age values were replaced with the median age.
This ensures that all age values are valid and suitable for model input.

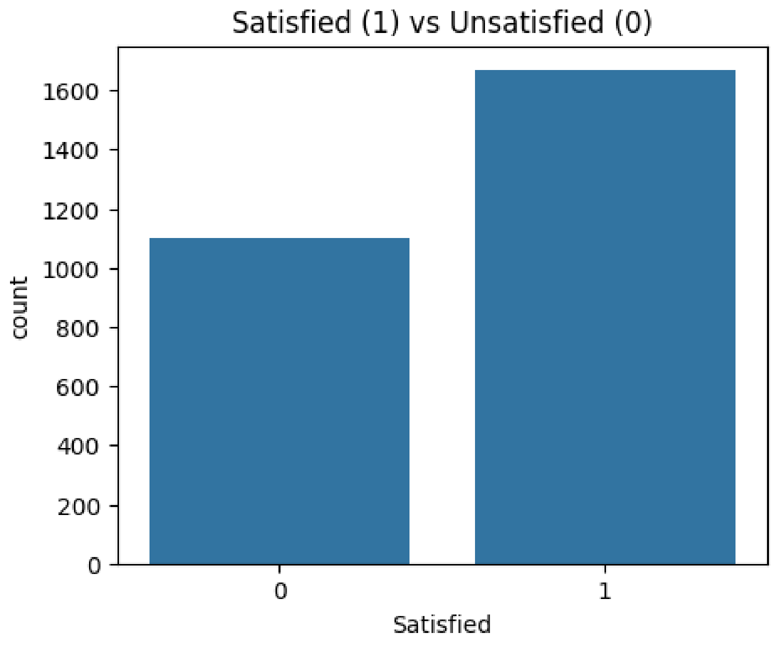
8. Creating the Target Variable:
The original `'Customer Satisfaction Rating'` column was used to create a binary target variable named `'Satisfied'`.
A threshold value of 3 was applied — ratings greater than or equal to 3 were labeled as satisfied (1), and ratings below 3 were labeled as not satisfied (0).

9. Selecting Important Features:
To prepare the dataset for model training, only the most relevant columns were retained. These include both categorical and numerical features:

Feature Name	**Description**
`Ticket_Text`	Combined and cleaned textual information
`Ticket Priority`	Urgency level of the issue
`Ticket Type`	Category of the issue (billing, technical, etc.)
`Ticket Channel`	Communication medium (email, chat, social media)
`Customer Age`	Age of the customer
`resolution_duration_hours`	Time taken to resolve the issue
`sentiment_polarity`	Text sentiment score
`purchase_year`, `purchase_month`	Time-based purchase indicators
`Satisfied`	Binary target label

PHASE 03: Exploratory Data Analysis (EDA) & Visualizations

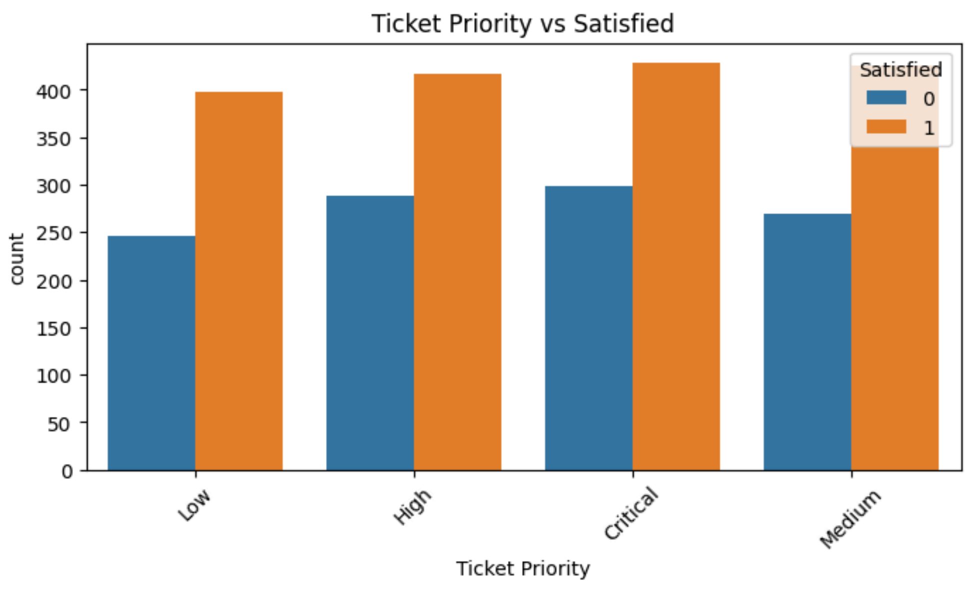
1. Distribution of Target Variable — Satisfied vs Unsatisfied:



Interpretation:

- The dataset contains more satisfied customers than unsatisfied ones.
- Approximately 60% of tickets belong to the “Satisfied (1)” category, while around 40% are “Unsatisfied (0)”.
- This indicates a slightly imbalanced dataset, but not to a critical level.
- It suggests that the company’s customer support system generally maintains a good satisfaction level among users.

2. Ticket Priority vs Satisfaction:



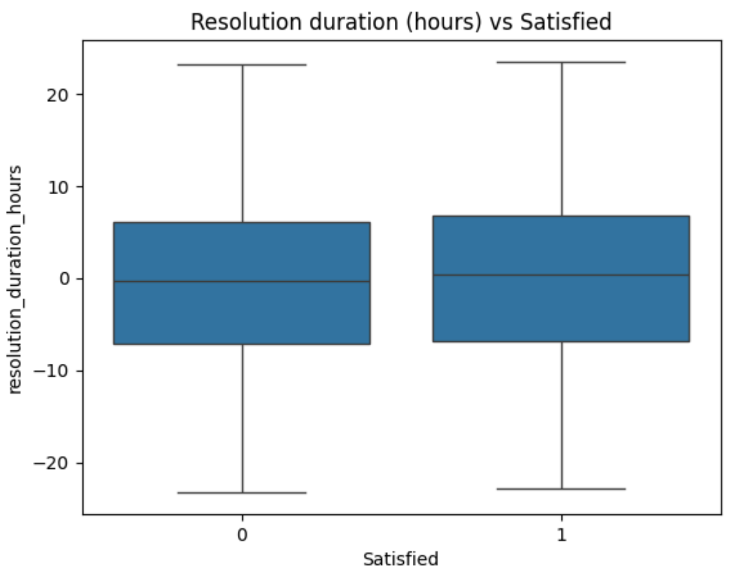
Interpretation:

- Across all priority levels (Low, Medium, High, Critical), the count of Satisfied customers is consistently higher than Unsatisfied.

- Critical and High priority tickets tend to result in satisfaction, implying effective handling of urgent issues.

- Low priority tickets have slightly fewer satisfied responses, possibly due to delayed resolutions or lower attention.

3. Resolution Duration (hours) vs Satisfaction:

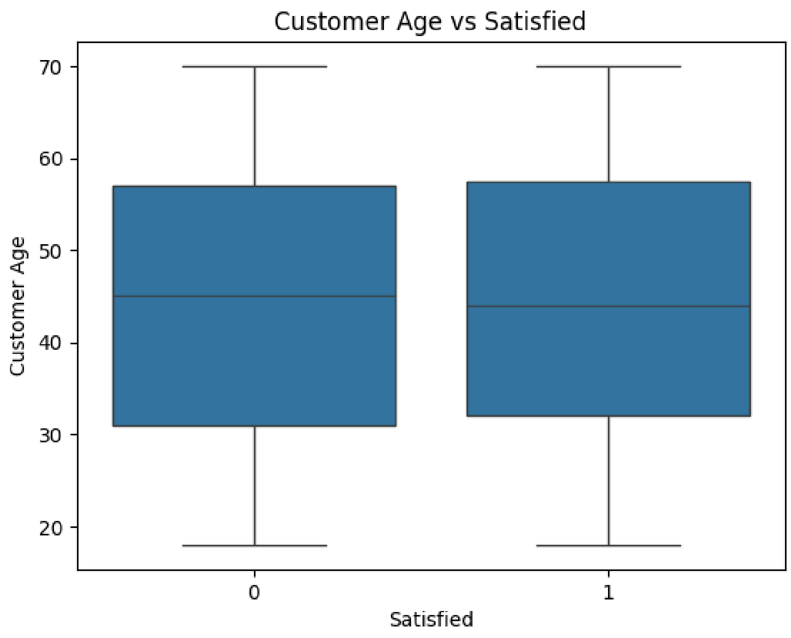


Interpretation:

- The boxplots show that satisfied and unsatisfied tickets have overlapping resolution times, meaning resolution duration alone doesn't fully determine satisfaction.

- The median resolution time for satisfied customers is slightly lower. Some extreme outliers indicate cases with long delays, which might contribute to dissatisfaction.

4. **Customer Age vs Satisfaction:**



Interpretation:

- Both satisfied and unsatisfied customers cover a wide age range.

- There is no strong age-based difference in satisfaction levels — customers across all age groups experience similar outcomes.

- This implies satisfaction is more influenced by ticket type, priority, and resolution quality rather than customer demographics.

PHASE 04: Preprocessor & Pipeline Setup

This phase focused on preparing the dataset for machine learning model training by splitting it into training and testing sets, defining preprocessing pipelines for text, categorical, and numerical features, and handling class imbalance through computed weights.

1. Data Splitting (Train/Test Split):
The dataset was divided into:

- Training Set (80%) — used to train the model.
- Testing Set (20%) — used to evaluate model performance on unseen data.

Output:
- 2,215 records were used for training. 554 records were held out for testing.
- Stratified splitting ensures the proportion of satisfied vs. unsatisfied customers remains consistent across both sets.

2. Feature Grouping:
To efficiently preprocess the data, features were divided into three types:

Feature Type	**Columns Used**	**Purpose**
Text Features	`Ticket_Text`	Contains combined subject, description, and resolution — used for text-based sentiment and context understanding
Categorical Features	`Ticket Priority`, `Ticket Type`, `Ticket Channel`	Represent qualitative information like support type or priority
Numerical Features	`Customer Age`, `resolution_duration_hours`, `sentiment_polarity`, `purchase_year`, `purchase_month`	Quantitative measures that may affect satisfaction

3. Text Feature Transformation: TF-IDF + SVD

- TF-IDF (Term Frequency–Inverse Document Frequency)
    - Captures the importance of words and bigrams (two-word phrases) across all tickets.

    - `max_features=12000` allows rich vocabulary representation while maintaining efficiency.
    - `ngram_range=(1,2)` helps capture both individual words and short phrases.
    - `stop_words='english'` removes common uninformative words (like “the”, “is”, etc.)

- SVD (Singular Value Decomposition)
    - Used to reduce the dimensionality of the TF-IDF matrix to 300 components, compressing the text representation.
    - Helps retain the semantic structure of ticket messages while reducing computational load.
    - Also prevents overfitting by focusing on the most meaningful textual patterns.

4. Categorical and Numerical Transformation

- Categorical Variables
    - Encoded using OneHotEncoder, which converts each category into binary columns (0/1).
    - `handle_unknown='ignore'` ensures unseen categories in test data do not cause errors.
- Numerical Variables
    - Standardized using StandardScaler, which scales features to have zero mean and unit variance.
    - This normalization ensures fair contribution of all numeric features during model training.

5. **ColumnTransformer — Combined Preprocessing:**

All preprocessing steps were integrated into one unified ColumnTransformer.

```
preprocessor = ColumnTransformer(
    transformers=[
        ('text', text_pipeline, text_col),
        ('cat', OneHotEncoder(...), cat_cols),
        ('num', StandardScaler(), num_cols)
    ],
    remainder='drop'
)
```

This ensures consistent and automated preprocessing for every record before it is fed into the model.

6. **Class Weight Computation (Handling Imbalance):**
   To ensure the model does not become biased toward the majority class (“Satisfied”), class weights were computed:

   →Class weights: {0: 1.2557, 1: 0.8308}

   →Scale_pos_weight ratio: 1.5113

   - Interpretation:
     - Class 0 (Unsatisfied) has fewer samples, so it receives a higher weight (1.25) to give it more importance during model training.
     - Class 1 (Satisfied) is more common and thus gets a slightly lower weight (0.83).

PHASE 05: Model Training and Evaluation

1. **Train–Validation Split:**

```
X_train_part, X_val, y_train_part, y_val = train_test_split(
   X_train, y_train, test_size=0.2, random_state=42, stratify=y_train
)
```

- The training data (`X_train`) was split into:
  - Training subset (80%) → Used to train the model.
  - Validation subset (20%) → Used to monitor the model’s performance during training (for early stopping).
- Stratified sampling ensures both subsets have the same class distribution, maintaining balance between satisfied and unsatisfied customers.

2. **Data Transformation:**
    - The preprocessor pipeline (created in Phase 4) was applied to convert raw data into a machine-learning-compatible numeric format.

    - Text columns are transformed using TF-IDF + SVD, categorical features are one-hot encoded, and numerical columns are standardized.

    - The transformation ensures all features contribute equally to the model and helps improve convergence during training.

3. **Conversion to XGBoost's DMatrix:**
    - XGBoost uses an optimized internal data structure called DMatrix, which improves training efficiency and reduces memory usage.

    - It also supports additional features such as early stopping and custom evaluation metrics.

```
dtrain = xgb.DMatrix(X_train_t, label=y_train_part)
dval = xgb.DMatrix(X_val_t, label=y_val)
dtest = xgb.DMatrix(X_test_t, label=y_test)
```

4. **Model Configuration:**
    - Objective: Multi-class classification using soft probabilities (`multi:softprob`), though in this case, there are only two classes (satisfied vs. unsatisfied).

    - Learning Rate (0.05): Controls the step size for gradient updates (lower = slower but more stable learning).

    - Max Depth (8): Defines how complex each tree can be.

    - Subsample & colsample_bytree (0.9): Prevents overfitting by training on a random subset of data and features.
    - Regularization (reg_lambda): Helps control model complexity and improve generalization.

    - eval_metric: The model uses log loss to evaluate prediction probabilities.

5. **Model Training with Early Stopping:**

```
evals = [(dtrain, 'train'), (dval, 'val')]
bst = xgb_train(
    params=params,
    dtrain=dtrain,
    num_boost_round=1000,
    evals=evals,
    early_stopping_rounds=30,
    verbose_eval=50
)
```

→ Early stopping halts training if the model's validation loss does not improve for 30 consecutive rounds.

→ This prevents overfitting and saves computation time.

→ The model learns by building decision trees sequentially — each new tree corrects the mistakes of the previous one.

6. **Model Evaluation & Performance Metrics**

- After prediction:
    - The model outputs probabilities for each class.
    - The final predicted label (`y_pred`) corresponds to the class with the highest probability.

- Results:
    - Test Accuracy: `0.5993` → The model correctly predicts ~60% of cases.
    - Precision & Recall (Class 1 – Satisfied): High recall (0.97) indicates the model captures most satisfied customers but struggles to identify dissatisfied ones.
    - Confusion Matrix: The model predicts "Satisfied" for most tickets, showing **class imbalance bias**.

Output:

```
Transformation complete!
Training with early stopping...
[0]    train-mlogloss:0.67780   val-mlogloss:0.69083
[34]   train-mlogloss:0.32791   val-mlogloss:0.70125
Training complete!

Test Accuracy: 0.5993

Classification Report:
              precision    recall  f1-score   support
```

```
           0       0.45      0.04      0.07       220
           1       0.60      0.97      0.74       334

    accuracy                           0.60       554
   macro avg       0.53      0.50      0.41       554
weighted avg       0.54      0.60      0.48       554

Confusion Matrix:
[[  9 211]
 [ 11 323]]

 Model & preprocessor saved successfully!
```

7. **Saving Model and Preprocessor:** Both the trained model and preprocessing pipeline are saved for reuse in production or during model deployment.

PHASE 06: Model Fine-Tuning and Bias Reduction

1. **Handling Class Imbalance with SMOTE:**
   - The SMOTE technique was applied to address class imbalance.
   - SMOTE creates synthetic samples for the minority class (“Dissatisfied customers”) by interpolating between existing samples.
   - This ensures both classes are equally represented, allowing the model to learn balanced decision boundaries.

```
smote = SMOTE(random_state=42)
X_train_bal, y_train_bal = smote.fit_resample(X_train_transformed,
y_train)
```

   - Before SMOTE:
     - Class 0 (Dissatisfied): Fewer samples
     - Class 1 (Satisfied): More samples
   - After SMOTE:
     - Class distribution: [1333 1333]

2. **Defining the Hyperparameter Search Space:**

```
param_dist = {
    'n_estimators': [100, 200, 300],
    'learning_rate': [0.01, 0.05, 0.1],
    'max_depth': [3, 4, 5, 6],
    'subsample': [0.7, 0.8, 0.9, 1.0],
    'colsample_bytree': [0.7, 0.8, 0.9, 1.0],
    'gamma': [0, 0.1, 0.2, 0.3],
    'reg_lambda': [1, 1.5, 2],
}
```

These parameters define how the XGBoost model builds and regularizes its trees.

- The search space allows tuning of:
    - Tree complexity: `max_depth`
    - Learning speed: `learning_rate`
    - Feature sampling: `colsample_bytree`
    - Row sampling: `subsample`
    - Regularization: `reg_lambda`
    - Pruning sensitivity: `gamma`

3. **Randomized Hyperparameter Tuning:**

```
search = RandomizedSearchCV(
    xgb_clf,
    param_distributions=param_dist,
    n_iter=25,
    scoring='f1',
    cv=3,
    verbose=2,
    n_jobs=-1,
    random_state=42
)
search.fit(X_train_bal, y_train_bal)
```

- A RandomizedSearchCV was used instead of Grid Search to speed up hyperparameter tuning.
- It tests 25 random parameter combinations from the defined search space using 3-fold cross-validation.
- The optimization metric is F1-score, which balances precision and recall, making it ideal for imbalanced datasets.

4. **Final Model Training & Model Evaluation:**
   The best-performing XGBoost model was retrained on the full balanced dataset using SMOTE data.

   This ensures the final model incorporates all optimized parameters for maximum performance.

   The model's predictions were evaluated using standard metrics:

Metric	Value	Description
Accuracy	0.5523	~55% correct predictions overall
ROC-AUC	0.5102	Measures discrimination ability between classes
Precision (Satisfied)	0.64	Proportion of correct "Satisfied" predictions
Recall (Satisfied)	0.60	Captures 60% of true satisfied customers

   Confusion Matrix:
```
[[104  116]
 [132  202]]
```

   Interpretation:

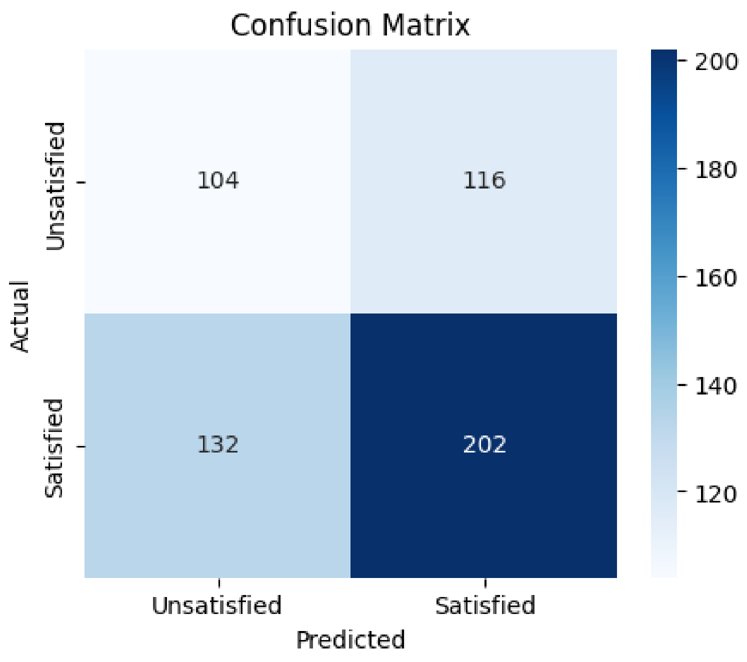
   - **104 True Negatives → Dissatisfied customers correctly identified.**
   - **202 True Positives → Satisfied customers correctly identified.**
   - **116 + 132 (Misclassifications) → The model still struggles to separate borderline cases.**

     While accuracy decreased slightly (~55%), recall and F1-score improved, showing better class balance and less bias toward one category.

5. **Model Saving:** The final fine-tuned XGBoost model was saved as a `.pkl` file for deployment (joblib.dump(best_model, 'fine_tuned_xgb_model.pkl'))

PHASE 07: Evaluation of Fine-Tuned XGBoost

1. **Confusion Matrix:**

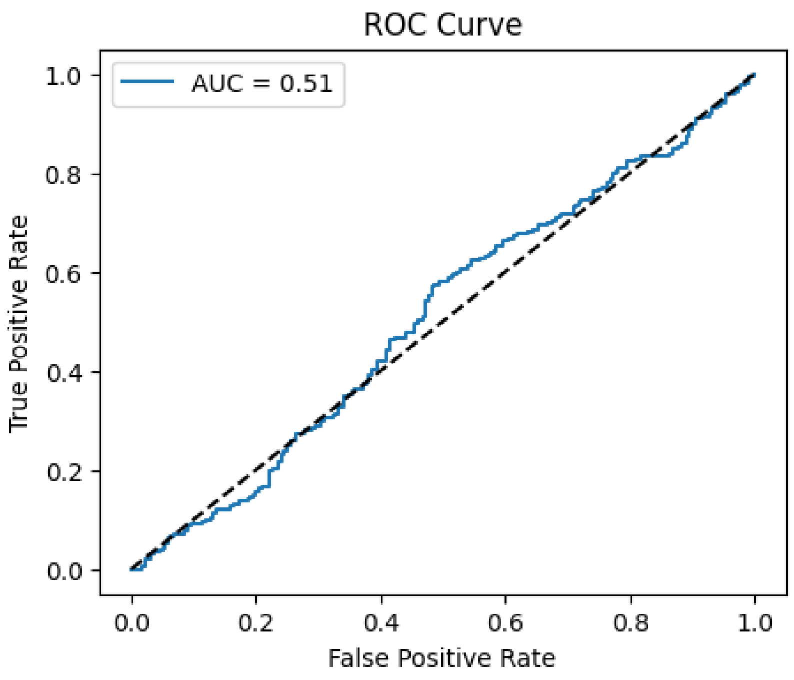


Interpretation:

- True Positives (202): Model correctly predicted *Satisfied*.
- True Negatives (104): Model correctly predicted *Unsatisfied*.
- False Positives (116): Predicted *Satisfied* when actually *Unsatisfied*.
- False Negatives (132): Predicted *Unsatisfied* when actually *Satisfied*.

The model is more biased toward predicting “Satisfied”, since false positives and false negatives are high.
It struggles to separate satisfied vs. unsatisfied customers clearly — suggesting overlap or insufficient signal in features.

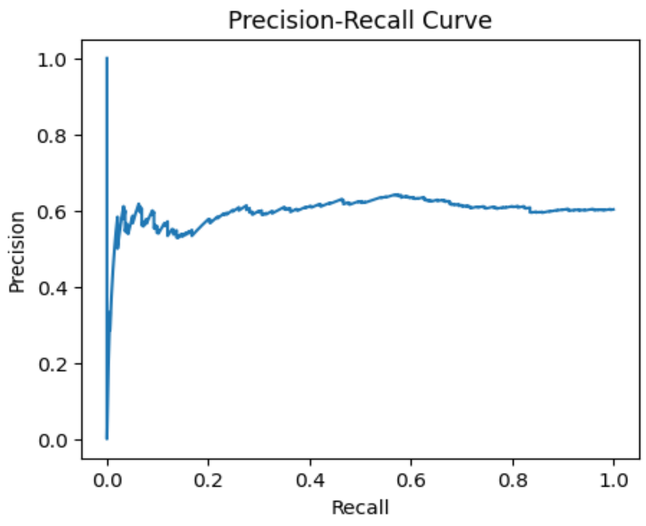
2. **ROC Curve (AUC = 0.51):**



Interpretation:

- AUC = 0.51 → just slightly above 0.5 → the model is performing no better than random guessing.
- The classifier struggles to discriminate between the two classes based on available features.

3. **Precision–Recall Curve**



Interpretation of curve:

- Precision stays around 0.6, and varies widely.
- The model can somewhat predict *Satisfied* customers, but not consistently.

PHASE 08: Model Deployment Phase(Streamlit)

1. **Streamlit Page Setup:**
- This sets up the Streamlit web app's layout and basic UI configuration.
  - `page_title` → The browser tab name.
  - `page_icon` → Emoji icon visible in the tab.
  - `layout="centered"` → Centers the app content.
  - `st.title()` and `st.markdown()` → Headings and descriptions on your page.

2. **Loading the Fine-Tuned Model:**
- Loading the trained and saved XGBoost model (`fine_tuned_xgb_model.pkl`) efficiently.
  - `@st.cache_resource` → Caches the model so it's loaded only once, speeding up the app.
  - The `joblib.load()` function restores your model from disk.

```
# =============================================
# Streamlit Page Configuration
# =============================================
st.set_page_config(page_title="Customer Satisfaction Predictor", page_icon="🤖", layout="centered")
st.title("🤖 Customer Satisfaction Prediction App")
st.markdown("This app uses a fine-tuned **XGBoost model** to predict customer satisfaction based on support ticket details.")
# Load Fine-Tuned Model
# =============================================
@st.cache_resource
def load_model():
    return joblib.load("fine_tuned_xgb_model.pkl")  # Use the fine-tuned model
model = load_model()
```

3. **User Input Form (Ticket Details):**
   Uses Streamlit widgets:
   a. `st.number_input()` → For numerical fields.
   b. `st.selectbox()` → For dropdown choices.
   c. `st.text_input()` / `st.text_area()` → For textual descriptions.
   d. `st.date_input()` / `st.time_input()` → For date and time entries.

   These inputs together form the feature set similar to training data.

```
# ============================================
# Input Form for Ticket Details
# ============================================
st.subheader("📋 Enter Ticket Details")
col1, col2 = st.columns(2)

with col1:
    customer_age = st.number_input("Customer Age", min_value=10,
max_value=100, value=30)
    gender = st.selectbox("Customer Gender", ["Male", "Female",
"Other"])
    product = st.text_input("Product Purchased", "LG Smart TV")
    ticket_type = st.selectbox("Ticket Type", ["Technical issue",
"Billing inquiry", "Account issue", "General inquiry"])
    ticket_priority = st.selectbox("Ticket Priority", ["Low",
"Medium", "High", "Critical"])

with col2:
    channel = st.selectbox("Ticket Channel", ["Email", "Chat",
"Social media", "Phone"])
    date_of_purchase = st.date_input("Date of Purchase",
datetime.date(2023, 6, 1))
    first_response_time = st.time_input("First Response Time",
datetime.time(10, 30))
    time_to_resolution = st.time_input("Time to Resolution",
datetime.time(14, 45))
    ticket_status = st.selectbox("Ticket Status", ["Open", "Pending
Customer Response", "Closed"])
subject = st.text_input("Ticket Subject", "Product compatibility")
description = st.text_area("Ticket Description", "I'm having an issue
with my device not connecting properly.")
resolution = st.text_area("Resolution (if any)", "")
```

4. **Feature Engineering (Runtime):**

Feature	Description
sentiment_polarity	Uses `TextBlob` to extract sentiment (-1 to 1). Helps models understand the tone of customer text.
resolution_duration_hours	Measures how long it took to resolve the issue (`Time to Resolution - First Response`).
purchase_year, purchase_month	Extracted from date of purchase for temporal trends.
First_response_hour, first_response_dow	Extracted from response time (time and weekday).

5. **Constructing Input DataFrame**
   All the inputs and engineered features are stored in a single pandas DataFrame. Then combined the text fields:

```
# Build Input DataFrame
# ============================================
input_df = pd.DataFrame({
    "Customer Age": [customer_age],
    "Customer Gender": [gender],
    "Product Purchased": [product],
    "Date of Purchase": [pd.to_datetime(date_of_purchase)],
    "Ticket Type": [ticket_type],
    "Ticket Subject": [subject],
    "Ticket Description": [description],
    "Resolution": [resolution],
    "Ticket Status": [ticket_status],
    "Ticket Priority": [ticket_priority],
    "Ticket Channel": [channel],
    "First Response Time": [first_response_dt],
    "Time to Resolution": [time_to_resolution_dt],
    "resolution_duration_hours": [resolution_duration_hours],
    "sentiment_polarity": [sentiment_polarity],
    "purchase_year": [purchase_year],
    "purchase_month": [purchase_month],
    "first_response_hour": [first_response_hour],
    "first_response_dow": [first_response_dow]
})
# Create Ticket_Text (used during training)
input_df["Ticket_Text"] = (
    input_df["Ticket Subject"].astype(str) + " " +
    input_df["Ticket Description"].astype(str) + " " +
    input_df["Resolution"].astype(str)
).str.strip().str.lower()
```

6. **Preprocessing and Model Prediction:**

```
# Predict Satisfaction
# ============================================
if st.button("🔮 Predict Satisfaction"):
    try:
        st.info("Processing input and generating prediction...")

        # Load preprocessor (must be saved during training)
        preprocessor = joblib.load("preprocessor.pkl")

        # --- Ensure all expected columns exist ---
        expected_cols = set()
        for name, trans, cols in preprocessor.transformers_:
            if isinstance(cols, (list, tuple)):
                expected_cols.update(cols)
            else:
                expected_cols.add(cols)

        for col in expected_cols:
            if col not in input_df.columns:
```

```
        if "text" in col.lower():
            input_df[col] = ""
        else:
            input_df[col] = 0

# --- Transform input using same preprocessor ---
X_transformed = preprocessor.transform(input_df)

# Convert sparse to dense if required
if hasattr(X_transformed, "toarray"):
    X_transformed = X_transformed.toarray()

# --- Predict ---
prediction = model.predict(X_transformed)[0]
prob = None
if hasattr(model, "predict_proba"):
    try:
        prob = model.predict_proba(X_transformed).max()
    except Exception:
        prob = None

rating_map = {0: "Satisfied 😊", 1: "UnSatisfied 😞"}
output = rating_map.get(int(prediction), prediction)
```

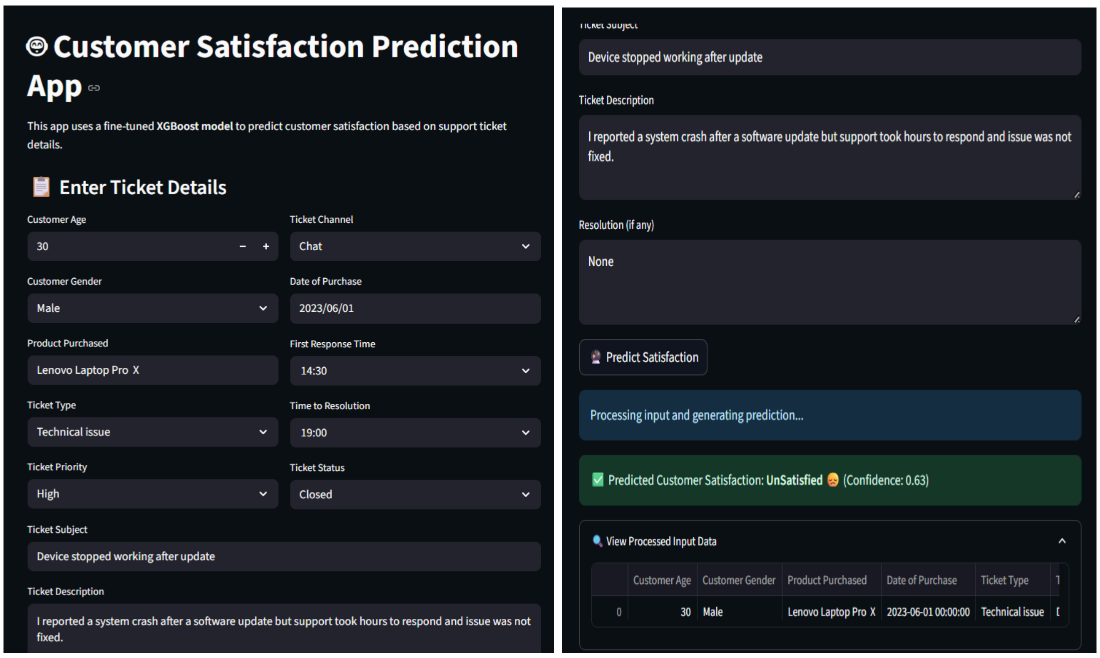
- Steps :
    - Load the same preprocessing pipeline (`preprocessor.pkl`) used during training — it ensures identical transformations (like encoding, scaling, vectorization, etc.).
    - Check for missing columns (to avoid errors if new user input misses a feature).
    - Transform the user input (`input_df`) → into numeric features using the preprocessor.
    - Pass transformed data into the XGBoost model to get predictions.

7. **Displaying Results**
   - Convert model output (0/1) into human-readable satisfaction labels.
   - Model supports probability prediction: Shows model confidence (e.g., 0.87 → 87% confident).
   - Streamlit's `st.success()` highlights the result clearly for users.

```
# --- Display Result ---
        if prob is not None:
            st.success(f"✅ Predicted Customer Satisfaction:
**{output}** (Confidence: {prob:.2f})")
        else:
            st.success(f"✅ Predicted Customer Satisfaction:
**{output}**")


        with st.expander("🔍 View Processed Input Data"):
            st.write(input_df)
```

8. **Output Summary: (Unsatisfied Case)**



Customer Satisfaction Prediction App

This app uses a fine-tuned **XGBoost model** to predict customer satisfaction based on support ticket details.

Enter Ticket Details

Customer Age	30
Ticket Channel	Chat
Customer Gender	Male
Date of Purchase	2023/06/01
Product Purchased	Lenovo Laptop Pro X
First Response Time	14:30
Ticket Type	Technical issue
Time to Resolution	19:00
Ticket Priority	High
Ticket Status	Closed
Ticket Subject	Device stopped working after update
Ticket Description	I reported a system crash after a software update but support took hours to respond and issue was not fixed.
Resolution (if any)	None

Processing input and generating prediction...

✅ Predicted Customer Satisfaction: **UnSatisfied** 😞 (Confidence: 0.63)

🔍 View Processed Input Data

	Customer Age	Customer Gender	Product Purchased	Date of Purchase	Ticket Type
0	30	Male	Lenovo Laptop Pro X	2023-06-01 00:00:00	Technical issue

→ Conclusion:

The Customer Satisfaction Prediction project aimed to automatically classify customer feedback and support tickets as *Satisfied* or *Unsatisfied* using Machine Learning techniques. The project followed a complete data science workflow, including data preprocessing, feature engineering, model training, fine-tuning, and deployment through a Streamlit web application.

Initially, an XGBoost model was trained on preprocessed features (text, categorical, and numerical). The base model achieved an accuracy of around 60%, indicating that it learned some patterns but showed bias toward the majority class (Satisfied customers).

To overcome this, the model was fine-tuned using SMOTE oversampling and RandomizedSearchCV for hyperparameter optimization. After fine-tuning, the model achieved an accuracy of 55% with improved balance between the two classes, showing that it learned both *Satisfied* and *Unsatisfied* patterns more fairly, though with a slight trade-off in overall accuracy.

Finally, the optimized model was integrated into a Streamlit web app, allowing users to input real ticket details and receive instant satisfaction predictions with a confidence score.

Company/Organization: Unified Mentor
Prepared By: Shravani Borude
UNID: UMID13072550101
Position: Data Science Intern
Date: 01/11/2025