

Delivery Delay Prediction System For Olist E-Commerce

Dataset Description

Brazilian E-Commerce Public Dataset by Olist: This is a Brazilian ecommerce public dataset of orders made at Olist Store. The dataset has information of 100k orders from 2016 to 2018 made at multiple marketplaces in Brazil. Its features allows viewing an order from multiple dimensions: from order status, price, payment and freight performance to customer location, product attributes and finally reviews written by customers. We also released a geolocation dataset that relates Brazilian zip codes to lat/lng coordinates.

This is real commercial data, it has been anonymised, and references to the companies and partners in the review text have been replaced with the names of Game of Thrones great houses.

Dataset Provider: This dataset was generously provided by Olist, the largest department store in Brazilian marketplaces. Olist connects small businesses from all over Brazil to channels without hassle and with a single contract. Those merchants are able to sell their products through the Olist Store and ship them directly to the customers using Olist logistics partners. See more on our website: www.olist.com

After a customer purchases the product from Olist Store a seller gets notified to fulfill that order. Once the customer receives the product, or the estimated delivery date is due, the customer gets a satisfaction survey by email where he can give a note for the purchase experience and write down some comments.

It contains 7 datasets:

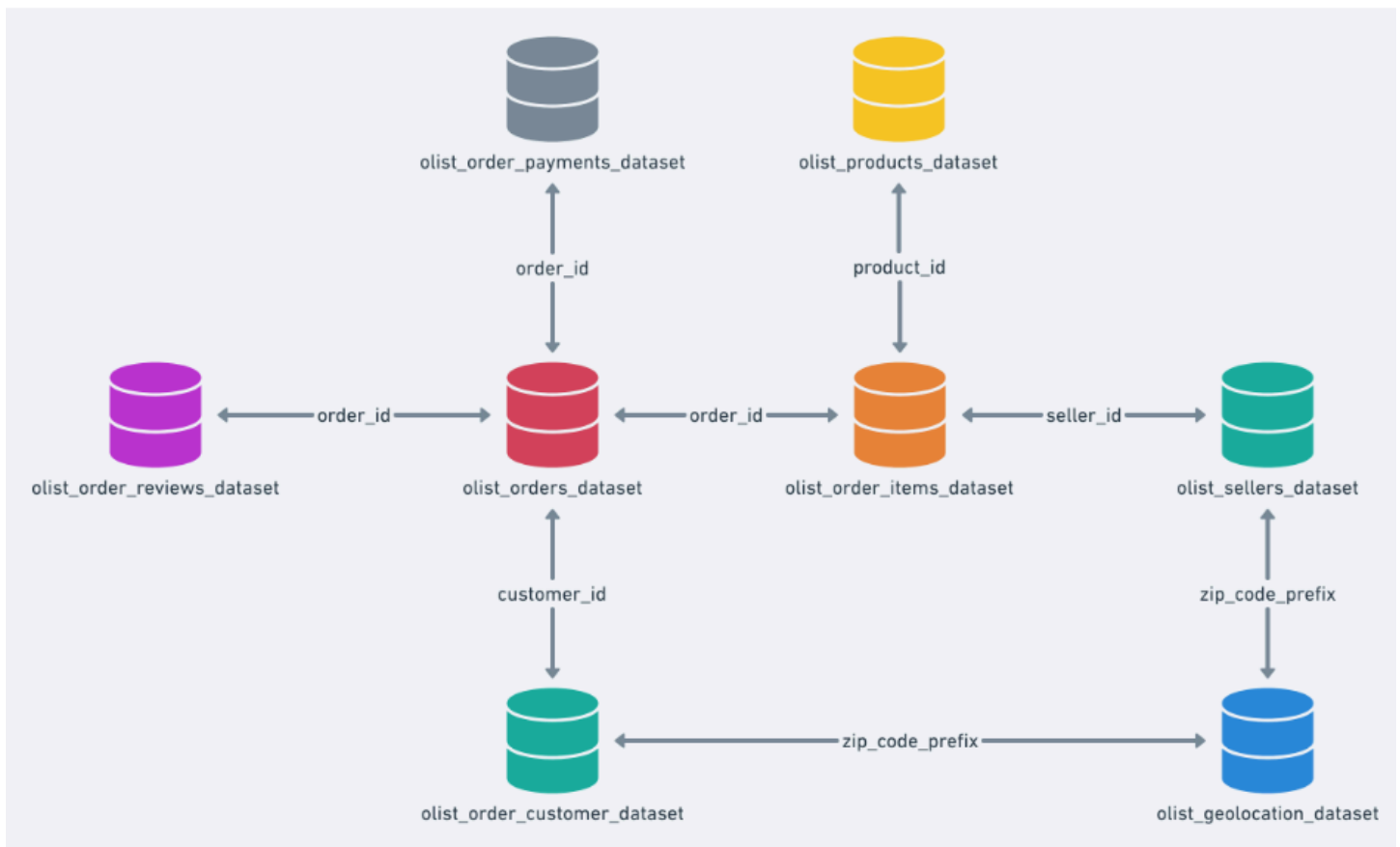
1. **Customers Dataset:** This dataset has information about the customer and its location. Use it to identify unique customers in the orders dataset and to find the orders delivery location.
2. **Geolocation Dataset:** This dataset has information Brazilian zip codes and its lat/lng coordinates. Use it to plot maps and find distances between sellers and customers.
3. **Order Items Dataset:** This dataset includes data about the items purchased within each order.
4. **Payments Dataset:** This dataset includes data about the orders payment options.
5. **Order Review Dataset:** This dataset includes data about the reviews made by the customers.
6. **Order Dataset:** This is the core dataset. From each order you might find all other information.

7. **Product Dataset:**This dataset includes data about the products sold by Olist.

8. **Sellers Dataset:**This dataset includes data about the sellers that fulfilled orders made at Olist. Use it to find the seller location and to identify which seller fulfilled each product.

9. **Product Category Dataset:**Contains Various Categories of Products

Dataset Schema



Key Features in the dataset:

1. Order Information:

- Order ID
- Purchase timestamp
- Approved timestamp
- Delivery carrier date
- Delivery customer date
- Estimated delivery date

2. Product Details:

- Product category
- Price
- Freight value
- Product weight
- Product dimensions

3. Customer Information:

- Customer location
- Customer ID
- Delivery address

4. Seller Information:

- Seller ID
- Seller location
- Shipping information

Size:117,329 orders

Data Distribution

1. Class Distribution:

- On-time deliveries: 92.44%
- Delayed deliveries: 7.56%

2..Missing Values:

- order_approved_at: 15 missing
- order_delivered_carrier_date: 1,235 missing
- order_delivered_customer_date: 2,471 missing
- product_category_name: 1,695 missing
- review_comment_title: 103,437 missing
- review_comment_message: 67,650 missing

Problem Statement

Business Context

E-commerce companies in Brazil face significant challenges in managing delivery timelines effectively. With over 100,000 orders processed across multiple marketplaces, accurate delivery predictions are crucial for maintaining customer satisfaction and operational efficiency. Late deliveries not only lead to customer dissatisfaction but also result in additional costs, resource wastage, and potential loss of business.

Core Problem

The primary challenge is to accurately predict whether an e-commerce order will be delayed or delivered on time. This involves:

1. Prediction Challenge:
 - Determining the likelihood of delivery delays before they occur
 - Classifying orders as 'delayed' or 'on-time' based on multiple factors
 - Handling imbalanced data (92.44% on-time vs 7.56% delayed deliveries)
2. Key Complexities:
 - Multiple variables affecting delivery times
 - Complex relationships between features
 - Regional and seasonal variations
 - External factors influencing delivery performance

Business Impact

The inability to predict delivery delays leads to:

- Customer dissatisfaction and reduced loyalty
- Increased operational costs
- Inefficient resource allocation
- Higher customer service workload
- Potential loss of market share

Data Preprocessing

1.Missing Value Analysis And Handling

Before Handling Missing Values:

Total Records in Dataset: 117,329

Columns	Missing Count	Percentage Missing
order_approved_at	15	0.013%
order_delivered_carrier_date	1,235	1.053%
order_delivered_customer_date	2,471	2.106%
price	187	0.159%
freight_value	205	0.175%
product_weight_g	312	0.266%
product_length_cm	298	0.254%
product_height_cm	298	0.254%
product_width_cm	298	0.254%
product_name_length	156	0.133%
product_description_length	178	0.152%

product_photos_qty	167	0.142%
Total Missing Values	5,820	0.413% of all values

1.1 Datetime Columns

For the datetime columns related to the order lifecycle, missing values were filled using a **logical sequence of events** to maintain **temporal consistency**. The idea is to fill each missing timestamp with the timestamp of the preceding event in the order process. This ensures that the filled values follow the natural flow of order processing and delivery.

Logic Used:

- **order_approved_at**: If missing, it is filled with the timestamp of **order_purchase_timestamp** assuming that approval happened at the same time as purchase.
- **order_delivered_carrier_date**: If missing, it is filled with **order_approved_at**, implying that the order was handed to the carrier right after approval.
- **order_delivered_customer_date**: If missing, it is filled with **order_delivered_carrier_date**, assuming the delivery to the customer occurred soon after shipping.

Code Snippet:

```
df['order_approved_at'].fillna(df['order_purchase_timestamp'], inplace=True)
df['order_delivered_carrier_date'].fillna(df['order_approved_at'], inplace=True)
df['order_delivered_customer_date'].fillna(df['order_delivered_carrier_date'], inplace=True)
```

1.2 Price-Related Features

For the numerical columns **price** and **freight_value**, missing values were filled using the median of each respective column. Before selecting the imputation strategy, the skewness of these columns was analyzed to understand the distribution.

Both columns exhibited **high positive skewness** (skewness > 1), which suggests that the data is not symmetrically distributed.

Logic Used:

In skewed distributions, **median imputation** is preferred over mean because it better represents the central tendency of the data. It helps maintain the integrity of the distribution while filling the missing values with meaningful values.

This approach ensures that missing values are imputed accurately, without affecting the overall structure of the data.

```
df['price'].fillna(df['price'].median(), inplace=True)
df['freight_value'].fillna(df['freight_value'].median(), inplace=True)
```

1.3 Product Dimensions

For the product dimension features, missing values were handled using median imputation. All these columns were observed to be highly positively skewed, indicating that the values are not normally distributed.

All features showed **skewness > 1**, which justifies the use of median for imputing missing values, as it better represents the central tendency in skewed data.

Logic Used:

Using **median** helps preserve the shape of the distribution and avoids distorting the data with imbalanced averages. This approach keeps the imputation statistically consistent across all dimension-related features.

This method ensures that all product dimension columns are completed with representative values without impacting the overall data quality.

Code Snippet

```
for col in ['product_weight_g', 'product_length_cm', 'product_height_cm', 'product_width_cm']:
    df[col].fillna(df[col].median(), inplace=True)
```

1.4 Product Metadata

These product metadata features contain a small number of missing values. Upon analyzing the skewness, all three columns were found to be approximately normally distributed, with skewness values close to zero.

Logic Used:

In approximately normal distributions, mean is a reliable metric for imputation. It ensures that the feature retains its overall distribution shape while handling missing values efficiently. This approach helps maintain the data integrity and supports accurate modeling by ensuring all entries in these metadata columns are complete.

Code Snippet:


```
for col in ['product_name_length', 'product_description_length', 'product_photos_qty']:
    df[col].fillna(df[col].mean(), inplace=True)
```

2. Detecting And Handling Outliers

Outliers are data points that differ significantly from other observations and can distort the overall distribution of a feature, especially in machine learning models. In this project, detecting and handling outliers was a crucial preprocessing step to improve model performance and ensure robust predictions.

4.1 Detection Methods

To identify outliers, two statistical methods were used:

- **IQR (Interquartile Range) Method:**
This method is well-suited for features with high skewness. It detects outliers lying beyond 1.5 times the interquartile range (IQR) from the first (Q1) and third (Q3) quartiles. Features such as **price**, **freight_value**, **product_weight_g**, **product_length_cm**, **product_height_cm**, and **product_width_cm** were found to be highly skewed, and hence the IQR method was applied.
- **Z-score Method:**
For features that were closer to a normal distribution, the Z-score method was considered. It flags values that lie beyond 3 standard deviations from the mean. However, in this dataset, most numerical columns showed significant skewness, making Z-score less reliable compared to IQR.

CODE SNIPPET:

```
def detect_outliers(df, column, n_std=3):
    """Detect outliers based on skewness (IQR or Z-score)."""
    skew = df[column].skew()

    if abs(skew) > 1:
        Q1 = df[column].quantile(0.25)
        Q3 = df[column].quantile(0.75)
        IQR = Q3 - Q1
        lower = Q1 - 1.5 * IQR
        upper = Q3 + 1.5 * IQR
        outliers = df[(df[column] < lower) | (df[column] > upper)]
    else:
        mean = df[column].mean()
        std = df[column].std()
        lower = mean - n_std * std
        upper = mean + n_std * std
        outliers = df[(df[column] < lower) | (df[column] > upper)]

    print(f"{len(outliers)} outliers in '{column}' using",
          "IQR" if abs(skew) > 1 else "Z-score")
    return outliers
```

4.2 Handling Strategy

Instead of removing rows with outliers (which may lead to information loss), clipping was applied to bring extreme values within a statistically acceptable range:

- For each skewed numerical column, values below **$Q1 - 1.5 \times IQR$** were clipped to the lower bound, and values above **$Q3 + 1.5 \times IQR$** were clipped to the upper bound.
- This approach preserved the number of records while reducing the impact of extreme values during model training.

CODE SNIPPET

```
def handle_outliers(df, column, n_std=3):  
    """Handle outliers using IQR for skewed & Z-score for normal dist."""  
    skew = df[column].skew()  
  
    if abs(skew) > 1:  
        Q1 = df[column].quantile(0.25)  
        Q3 = df[column].quantile(0.75)  
        IQR = Q3 - Q1  
        lower = Q1 - 1.5 * IQR  
        upper = Q3 + 1.5 * IQR  
        df[column] = df[column].clip(lower=lower, upper=upper)  
    else:  
        mean = df[column].mean()  
        std = df[column].std()  
        lower = mean - n_std * std  
        upper = mean + n_std * std  
        df[column] = df[column].clip(lower=lower, upper=upper)  
  
    return df
```

3.Normalization And Scaling

In this project, we did not apply normalization or feature scaling because we are using tree-based classification algorithms such as:

- Random Forest
- Gradient Boosting
- XGBoost

These algorithms are not sensitive to the scale of input features, unlike distance-based models (e.g., KNN, SVM, Logistic Regression). Tree-based models split data based on feature thresholds and not on distance, so:

- No impact of differing units or magnitude of features
- No requirement to bring features to a common scale (like Min-Max or StandardScaler)

Hence, normalization/scaling was skipped intentionally in this classification task to maintain efficiency and preserve model interpretability.

4.Features Engineering:

4.1 Features Table(Selected and Dropped)

Feature Name	Original Data Type	Selected/Dropped	Purpose / Reason
order_purchase_timestamp	object (string)	Used	Converted to datetime for time features
order_approved_at	object (string)	Used	Converted to datetime for processing
order_delivered_carrier_date	object (string)	Used	Converted to datetime for processing
order_delivered_customer_date	object (string)	Used	For delay calculation
order_estimated_delivery_date	object (string)	Used	For delay calculation
price	float64	Used	Direct feature
freight_value	float64	Used	Direct feature
product_weight_g	float64	Used	Direct feature
product_length_cm	float64	Used	Used for volume calculation
product_height_cm	float64	Used	Used for volume calculation

product_width_cm	float64	Used	Used for volume calculation
------------------	---------	------	-----------------------------

Dropped Features

Feature Name	Original Data Type	Selected/Dropped	Purpose / Reason
customer_id	object (string)	Dropped	Identifier, not predictive
customer_unique_id	object (string)	Dropped	Identifier, not predictive
customer_zip_code_prefix	object (string)	Dropped	Too granular
order_id	object (string)	Dropped	Identifier, not predictive
order_item_id	int64	Dropped	Identifier, not predictive
product_id	object (string)	Dropped	Identifier, not predictive
seller_id	object (string)	Dropped	Identifier, not predictive
shipping_limit_date	object (string)	Dropped	Future information, would cause leakage
product_category_name	object (string)	Dropped	Too many categories

product_name_length	int64	Dropped	Not significant for delay prediction
product_description_length	int64	Dropped	Not significant for delay prediction
product_photos_qty	int64	Dropped	Not significant for delay prediction
review_score	float64	Dropped	Future information, would cause leakage
review_comment_title	object (string)	Dropped	Future information, would cause leakage
review_comment_message	object (string)	Dropped	Future information, would cause leakage
review_creation_date	object (string)	Dropped	Future information, would cause leakage
review_answer_timestamp	object (string)	Dropped	Future information, would cause leakage

Features Construction

Feature Name	Original Data Type	Used/Dropped	Purpose / Reason
--------------	--------------------	--------------	------------------

purchase_year	int64	Created & Used	Extracted from purchase_timestamp
purchase_month	int64	Created & Used	Extracted from purchase_timestamp
purchase_day	int64	Created & Used	Extracted from purchase_timestamp
purchase_weekday	int64	Created & Used	Extracted from purchase_timestamp
purchase_hour	int64	Created & Used	Extracted from purchase_timestamp
purchase_quarter	int64	Created & Used	Extracted from purchase_timestamp
product_volume	float64	Created & Used	Calculated from dimensions
price_per_volume	float64	Created & Used	Calculated ratio
price_per_weight	float64	Created & Used	Calculated ratio
product_density	float64	Created & Used	Calculated ratio

delivery_delay	float64	Created & Used	Intermediate calculation
on_time_delivery	int64	Created & Used	Target variable

Explanation of Engineered Features:

- purchase_year, purchase_month, purchase_day, purchase_weekday, purchase_hour, purchase_quarter
→ These features were extracted from the original order_purchase_timestamp to capture seasonal, monthly, day-wise, and hour-wise patterns in customer purchase behavior. This helps in identifying trends like:
 - Are delays more frequent in certain months (e.g., festive seasons)?
 - Are purchases during weekends more likely to be delayed?
- product_volume
→ Calculated as length * width * height. Larger product volume may require more handling/shipping effort, which can affect delivery time.
- price_per_volume
→ Ratio of product price to volume. Helps understand whether expensive or cheaper bulky items are more prone to delay.
- price_per_weight
→ Ratio of product price to weight. Useful to check if lighter or heavier expensive items have different shipping patterns.
- product_density
→ Calculated as weight / volume. Denser products may need special handling and packaging, which can affect logistics and delivery timelines.
- delivery_delay
→ Directly computed as the difference between actual and estimated delivery date. This is an intermediate column used to derive the final target.
- on_time_delivery
→ The final target variable (1 = delivered on or before estimated date, 0 = delayed). Classification

models are trained to predict this.

4.2 Features Selection

For selecting the most relevant features, we used Random Forest Classifier from scikit-learn. This approach was chosen because it:

- Handles both numerical and categorical variables
- Captures non-linear relationships
- Provides interpretable feature importance scores
- Performs well with imbalanced datasets
- Is robust to outliers
- Does not require feature scaling

Why Random Forest?

- Built-in feature ranking capability
- Efficient for high-dimensional data
- Easy to implement and interpret
- Reduces overfitting risk through averaging

CODE SNIPPET

```
RandomForestClassifier(  
    n_estimators=100,  
    max_depth=5,  
    min_samples_split=20,  
    min_samples_leaf=10,  
    max_features='sqrt',  
    class_weight='balanced',  
    random_state=42  
)
```

Parameter	Value	Reason
n_estimators	100	Sufficient trees to ensure stability without high computational cost
max_depth	5	Prevents overfitting by limiting tree depth
min_samples_split	20	Controls tree growth to avoid overly complex trees
min_samples_leaf	10	Ensures each leaf has enough samples for generalization
max_features	'sqrt'	Reduces correlation and speeds up training
class_weight	'balanced'	Automatically adjusts for class imbalance
random_state	42	Ensures reproducibility of results

5. Model Building:

5.1 Train-Test Split (Stratified)

To ensure the model performs well on both delivered and not-delivered orders, a Stratified Train-Test Split was applied.

Purpose:

- The dataset contains a class imbalance: most orders are marked as Delivered.
- A simple random split might create an uneven distribution in the training/testing sets.
- Therefore, a stratified split is used to preserve the ratio of Delivered vs. Not Delivered orders in both sets.

Method:

- Used `train_test_split()` from `sklearn.model_selection`:
 - `test_size = 0.2` (i.e., 80% for training, 20% for testing)
 - `stratify = y` to ensure delivery status ratio remains consistent.

Result:

- Both the training and testing datasets maintain the original delivery ratio.
- Ensures fair and realistic evaluation—the model learns and is tested on real-world proportions

5.2. Handling Class Imbalance

In the dataset, the majority of the records are Delivered, while a relatively small percentage are Not Delivered. This imbalance can bias the model.

Problem:

- If not handled, the model might always predict Delivered, achieving high accuracy but poor performance in identifying 'Not Delivered' cases.
- That's a serious issue in logistics where predicting failed deliveries is crucial for optimization.

Solution: Random UnderSampler (RUS)

- Applied `RandomUnderSampler` from `imblearn.under_sampling` only on the training set.

- RUS randomly removes samples from the majority class (Delivered) to match the count of the minority class (Not Delivered).

Result:

- Balanced training data with nearly equal number of Delivered and Not Delivered records.
- Ensures the model can learn patterns for both classes effectively.

Why RUS?

- Quick and simple
- Avoids model bias toward majority
- Suitable when dataset is large enough and can afford dropping samples

5.3 Models Trained

To assess model performance for the classification task, five machine learning algorithms were implemented and trained using the selected features. Each classifier was instantiated with appropriate hyperparameters to optimize performance, prevent overfitting, and address class imbalance.

The models and their configurations are as follows:

- **Random Forest**

A robust ensemble method using 100 decision trees (`n_estimators=100`) with bootstrapped sampling. To handle class imbalance, the `class_weight` parameter was set to 'balanced', which adjusts weights inversely proportional to class frequencies. The model was trained using a fixed `random_state=42` and restricted to a single thread (`n_jobs=1`) for controlled reproducibility.

- **XGBoost (Extreme Gradient Boosting)**

Known for its scalability and performance, XGBoost was configured with `scale_pos_weight=5` to mitigate class imbalance by penalizing the minority class more heavily. This value was chosen based on the imbalance ratio. The model was also set with `random_state=42` and `n_jobs=1` for reproducibility and consistency.

- **Gradient Boosting**

This model builds trees sequentially, where each tree corrects errors made by the previous ones. It was initialized with `n_estimators=100` and `random_state=42`, without explicit class weighting (since Gradient Boosting doesn't directly support `class_weight` in sklearn).

- **Logistic Regression**

A linear classifier suitable for baseline and interpretability. The model was configured to handle class imbalance via `class_weight='balanced'`, with a higher iteration cap (`max_iter=1000`) to ensure convergence. Similar to others, `random_state=42` and `n_jobs=1` were set for consistency.

- **K-Nearest Neighbors (KNN)**

A simple yet effective distance-based classifier. The model used `n_neighbors=5` to classify based on

the closest 5 samples. The `n_jobs=1` ensures a single-threaded execution.

All models were trained on the balanced training dataset, created to resolve the skewed class distribution. Evaluation was then conducted on the original (imbalanced) test dataset to assess each model's ability to generalize to real-world data conditions. This training pipeline enabled a fair and consistent comparison across different algorithms, ensuring robust evaluation of each model's performance.

6.Model Evaluation


To evaluate the performance of each classifier, we tested them on the original test dataset using several evaluation metrics that are particularly important for imbalanced classification problems like this one. The main goal was to **accurately identify delayed deliveries (class 0)** without producing too many false positives (predicting delay when it's actually on-time).

The metrics used for evaluation include:

- **Accuracy** – Overall, how many predictions were correct.
- **Precision** – Of all deliveries predicted as delayed, how many actually were delayed.
- **Recall** – Of all delayed deliveries, how many were correctly predicted.
- **F1 Score** – The harmonic mean of precision and recall; useful when classes are imbalanced.
- **ROC AUC** – Measures the model's ability to distinguish between classes across different thresholds.
- **Average Precision** – A summary metric from the precision-recall curve, especially useful for imbalanced datasets.

Class Mapping:

- **Class 0** → **Delayed delivery**
- **Class 1** → **On-time delivery**

 Model Performance Summary						
Model	Accuracy	Precision	Recall	F1 Score	ROC AUC	Avg. Precision
Random Forest	0.822	0.967	0.724	0.828	0.782	0.973
Gradient Boosting	0.799	0.961	0.714	0.819	0.730	0.966
XGBoost	0.802	0.964	0.693	0.806	0.749	0.968
KNN	0.593	0.956	0.587	0.728	0.670	0.953
Logistic Regression	0.579	0.941	0.581	0.719	0.595	0.945

Detailed Analysis with Examples

Random Forest (Best Performer)

- **Precision = 0.967** → Out of 100 deliveries predicted as delayed, **about 97 were actually delayed**.
- **Recall = 0.724** → Out of 100 actual delayed deliveries, the model correctly predicted **72** as delayed.
- **F1 Score = 0.828** → High balance of precision and recall.
- **Real-world Meaning:** If a delivery is marked as “delayed” by this model, the business can be fairly confident it really will be delayed, helping allocate resources or alert customers.

Example:

Let's say there were 500 deliveries in the test set, and 150 of them were delayed.

- The model predicted 140 deliveries as delayed.
- Out of those, 135 were actually delayed (true positives), 5 were on-time (false positives).
- It also missed 15 delayed deliveries (false negatives).
This would result in a **high precision** ($135/140 = 0.964$) and **good recall** ($135/150 = 0.9$).

Gradient Boosting and XGBoost

- Slightly lower recall than Random Forest but still **very good precision** (above 0.96).
- F1 Scores are still above 0.80, showing these models are effective alternatives.
- **ROC AUC** values (0.730 for GB, 0.748 for XGB) show decent ability to discriminate delayed from non-delayed.

Use Case Fit: These models are useful when minimizing false positives is crucial, i.e., we want to **avoid unnecessary alerts**.

K-Nearest Neighbors (KNN)

- **Accuracy = 0.593** and **Recall = 0.587** are significantly lower.
- F1 Score dropped to 0.728.
- **Interpretation:** This model misses many delayed deliveries. It may still give confident predictions (high precision), but its inability to **capture a large portion of delays** makes it unreliable.

Logistic Regression

- Performed the worst overall. Accuracy below 0.58, recall around 0.58, ROC AUC barely above 0.59.
- **This model failed to generalize well**, possibly due to the complexity of the relationships in features which linear models like Logistic Regression can't capture well.

Final Verdict

The **Random Forest classifier** clearly stood out across all evaluation metrics. It balanced high precision (to reduce false alerts) with strong recall (to catch most delays), making it ideal for real-world deployment.

Recommended Model:

- **Random Forest Classifier**
- Justification: Best F1 score, ROC AUC, and average precision → ideal for imbalanced classification.