ML Project: Food\_Delivery\_time (mins)

**Problem Statement:**

**Objective:**  
To predict the delivery time for food orders based on various factors such as distance, weather, traffic level, time of day, vehicle type, preparation time, and the courier's experience.

**Context:**  
In the food delivery industry, delivery time is a crucial factor for both customers and delivery companies. Accurate prediction of delivery time helps in managing customer expectations, optimizing delivery routes, and improving operational efficiency. A reliable prediction model can aid in better planning, resource allocation, and overall performance.

**Problem Description:**  
Given a dataset containing details of food deliveries, including:

* **Distance (km)**: The distance between the restaurant and the delivery location.
* **Weather conditions**: The weather on the delivery day (e.g., clear, rainy, foggy, snowy).
* **Traffic Level**: The level of traffic during the delivery (e.g., low, medium, high).
* **Time of Day**: The time at which the delivery is made (e.g., morning, afternoon, evening, night).
* **Vehicle Type**: The type of vehicle used for delivery (e.g., bike, scooter, car).
* **Preparation Time**: The time taken to prepare the food at the restaurant.
* **Courier Experience**: The experience level of the courier (measured in years).

The **goal** of this project is to develop a machine learning model that predicts the **delivery time** (in minutes) for a given food order based on the above features. The model should be able to handle the various factors that influence delivery time and provide accurate predictions for new orders.

**Key Challenges:**

* **Non-linear relationships**: The relationship between delivery time and various features is complex, especially considering factors like weather, traffic, and vehicle type.
* **Feature Interactions**: Some features may have a combined effect on delivery time (e.g., bad weather + high traffic can drastically increase delivery time).
* **Data Quality**: Ensuring the data is clean and properly preprocessed (handling missing values, encoding categorical variables, etc.).

**Expected Outcome:**  
A trained machine learning model capable of accurately predicting the delivery time for new food delivery orders, helping businesses optimize delivery processes, reduce delays, and improve customer satisfaction.

**Evaluation Metric:**  
The model will be evaluated based on metrics such as **Mean Absolute Error (MAE)**, **Mean Squared Error (MSE)**, or **Root Mean Squared Error (RMSE)** to assess the accuracy of the predicted delivery times compared to the actual delivery times.

In your dataset, you have both **numerical** and **categorical** features. Encoding categorical variables is an essential part of preparing your data for machine learning models, as most models require numerical inputs.

Let's walk through the types of encoding and the most suitable ones for your dataset.

**1. One-Hot Encoding (OHE)**

**Best suited for:** **Nominal** categorical features with no inherent order or hierarchy.

**Use cases in your dataset:**

* **Weather** (e.g., Clear, Rainy, Foggy, Snowy)
* **Time\_of\_Day** (e.g., Morning, Afternoon, Evening, Night)
* **Vehicle\_Type** (e.g., Bike, Scooter, Car)

These features are **nominal** (i.e., categories without a natural ordering), so **One-Hot Encoding** is the best choice. It creates a binary (0 or 1) column for each unique category in the feature.

**Why OHE?**

* **Preserves uniqueness** of each category.
* Avoids assigning arbitrary numeric values that might suggest an incorrect ordering.

**Example (for Weather):**

plaintext

Copy code

Weather

Clear → [1, 0, 0, 0]

Rainy → [0, 1, 0, 0]

Foggy → [0, 0, 1, 0]

Snowy → [0, 0, 0, 1]

You can use **pandas.get\_dummies()** or **sklearn.preprocessing.OneHotEncoder** to apply this encoding.

**2. Label Encoding**

**Best suited for:** **Ordinal** categorical features, where the categories have a natural ordering or ranking.

**Use cases in your dataset:**

* **Traffic\_Level** (e.g., Low, Medium, High)  
  Here, there is a **natural order** (Low < Medium < High), so label encoding is appropriate. By assigning an integer to each level, the model can understand the relative order of traffic levels.

**Why Label Encoding?**

* It transforms each category into a number, which preserves the inherent order. For example:
  + Low → 0
  + Medium → 1
  + High → 2

**Example (for Traffic\_Level):**

plaintext

Copy code

Traffic\_Level

Low → 0

Medium → 1

High → 2

You can use **sklearn.preprocessing.LabelEncoder()** for this.

**3. Ordinal Encoding**

**Best suited for:** **Ordinal** categorical features with a defined order, similar to label encoding, but it's specifically designed for ordered categories.

**Use cases in your dataset:**

* **Traffic\_Level** (e.g., Low, Medium, High) can be treated as ordinal because it has a clear order, like **Label Encoding**. However, **Ordinal Encoding** is often used when you have a predefined set of categories with a specific ranking.

In practice, **Label Encoding** and **Ordinal Encoding** are very similar, but **Ordinal Encoding** explicitly indicates that the data has a specific order, and it can be used if the model will treat the values as ordered.

**4. Frequency or Count Encoding**

**Best suited for:** When there are **high-cardinality** categorical variables (i.e., many unique categories).

**Use cases in your dataset:**

* This encoding might not be as applicable here because the categorical features (like **Weather**, **Time\_of\_Day**, and **Vehicle\_Type**) seem to have relatively low cardinality. However, if you had a feature like **Courier\_ID** or any feature with many unique values, **Frequency Encoding** might be useful.

**Why Frequency Encoding?**

* It encodes categories based on the frequency or the count of occurrences in the dataset, so it helps to reduce the dimensionality when there are too many categories. But it might not work well for categorical features where each category carries specific meaning.

**5. Target Encoding (Mean Encoding)**

**Best suited for:** When a categorical feature is highly correlated with the target and you want to encode it based on the mean of the target variable for each category.

**Use cases in your dataset:**

* For example, if you found that the **Vehicle\_Type** or **Time\_of\_Day** has a strong relationship with **Delivery\_Time\_min**, you might use Target Encoding. This technique replaces each category with the mean target value for that category.

**Why Target Encoding?**

* It can be helpful if certain categories are highly predictive of the target, as it will help the model recognize patterns in the data. However, **Target Encoding** needs to be done carefully to avoid data leakage or overfitting.
* **Which Encoding to Use for Your Dataset?**

Based on the columns in your dataset, here's a breakdown of the most suitable encoding methods:

| **Feature** | **Type** | **Recommended Encoding** | **Reason** |
| --- | --- | --- | --- |
| **Weather** | Nominal | One-Hot Encoding | Categories have no order, so separate columns are best. |
| **Traffic\_Level** | Ordinal | Label Encoding or Ordinal Encoding | Natural order (Low < Medium < High) suggests numerical encoding. |
| **Time\_of\_Day** | Nominal | One-Hot Encoding | No natural order. |
| **Vehicle\_Type** | Nominal | One-Hot Encoding | No natural order. |
| **Preparation\_Time\_min** | Numerical | No encoding needed | Already a continuous variable. |
| **Courier\_Experience\_yrs** | Numerical | No encoding needed | Already a continuous variable. |

**Expected Outcome:**  
A trained machine learning model capable of accurately predicting the delivery time for new food delivery orders, helping businesses optimize delivery processes, reduce delays, and improve customer satisfaction.

The model will be evaluated based on metrics such as **Mean Absolute Error (MAE)**, **Mean Squared Error (MSE)**, or **Root Mean Squared Error (RMSE)** to assess the accuracy of the predicted delivery times compared to the actual delivery times.

* **Linear Regression:**

 **Heteroscedasticity** occurs when the variance of errors is not constant across different values of the independent variables. WLS addresses this by assigning more weight to observations with lower error variance and less weight to observations with higher error variance.

 By doing this, WLS helps to ensure that the model does not overly focus on data points with high variance, which would otherwise lead to inefficient or biased estimates.

**Weighted Least Squares (WLS)** is a regression technique used to address **heteroscedasticity** in the data, which occurs when the variance of the residuals is not constant across all levels of the independent variables. In other words, when the error terms exhibit non-constant variance, WLS can help by assigning weights to the observations, giving less importance to those with higher variance and more importance to those with lower variance.

 **Assign weights**: You need to calculate or specify weights that represent the inverse of the variance of the observations.

 **Fit the model**: Use the weighted least squares to fit the regression model, where the weights reduce the influence of higher-variance observations.

 **Interpret the results**: After fitting the model, you can assess the performance and make predictions.

* **Ridge Regression (L2 regularization)**: The model tries to minimize the sum of squared errors while also minimizing the sum of the squared coefficients. This results in smaller coefficients, which helps prevent overfitting.
* **Lasso Regression (L1 regularization)**: Similar to Ridge but with L1 regularization, which can lead to some coefficients becoming exactly zero. This makes Lasso useful for **feature selection**, as it effectively removes unimportant features from the model.
* **alpha Parameter**: Both Ridge and Lasso use alpha to control the strength of the regularization. A very large alpha will result in a highly regularized model with very small coefficients, possibly leading to underfitting, while a very small alpha might not regularize enough, leading to overfitting.

By using Ridge and Lasso regression, you can reduce overfitting in linear regression models, handle multicollinearity, and perform feature selection (in the case of Lasso).

* **Support vector machine (regression):**

**SVM for regression** using **GridSearchCV** or **RandomizedSearchCV** to find the optimal configuration. Below are the important hyperparameters to focus on:

1. **C (Regularization Parameter)**: Controls the trade-off between having a smooth decision boundary and classifying the training points correctly. A small C makes the decision boundary smooth, while a large C tries to classify all training points correctly, which can lead to overfitting.
2. **Epsilon (ε)**: Defines a margin of tolerance where no penalty is given for errors. A higher value of epsilon leads to a simpler model (potential underfitting), while a lower value of epsilon may lead to a more complex model (potential overfitting).
3. **Kernel**: Defines the function used to map the input data into a higher-dimensional space. The common kernels are:
   * **Linear**: For linearly separable data.
   * **RBF** (Radial Basis Function): Suitable for non-linear data.
   * **Polynomial**: Suitable for data where polynomial relationships exist.

* **Decision tree:**

**Key Hyperparameters to Tune for Preventing Overfitting in Decision Trees:**

1. **max\_depth**: Limits the depth of the tree. A very deep tree can easily overfit the training data by learning too much about the noise. Limiting the depth prevents the tree from growing too complex.
2. **min\_samples\_split**: The minimum number of samples required to split an internal node. Increasing this number will prevent the model from creating nodes based on very few samples, thus preventing overfitting.
3. **min\_samples\_leaf**: The minimum number of samples required to be at a leaf node. A larger number of samples in each leaf will ensure that each prediction is based on a broader sample of data, reducing the model's ability to overfit.
4. **max\_features**: The maximum number of features to consider when splitting a node. Limiting the number of features can help make the tree less sensitive to noise in the data.
5. **max\_leaf\_nodes**: The maximum number of leaf nodes in the tree. This hyperparameter allows controlling the tree's complexity.
6. **criterion**: The function to measure the quality of a split. "gini" for the Gini impurity and "entropy" for the information gain.
7.  **param\_grid**: This is the grid of hyperparameters to search over. We define different values for max\_depth, min\_samples\_split, min\_samples\_leaf, max\_features, etc.
8.  **GridSearchCV**: This class performs an exhaustive search over the parameter grid using cross-validation (cv=5 in this case).
9.  **scoring='neg\_mean\_squared\_error'**: We use the negative mean squared error as the scoring metric to evaluate the performance of the model during the search.
10.  **best\_params\_**: After fitting the model, we can extract the best hyperparameters that were found during the grid search.
11.  **Model evaluation**: After finding the best model, we evaluate it on the training and testing data by calculating R² and RMSE.

* **KNN regression:**

overfitting in KNN regression:

**1. Increase the Number of Neighbors (k):**

* **Problem**: When k is small (e.g., 1 or 2), the model is too sensitive to the noise in the data, resulting in overfitting.
* **Solution**: Increase the value of k. A higher value of k means the model will average over a larger number of neighbors, reducing the sensitivity to noise.
* **How to choose k**: A common approach is to try different values of k and evaluate the model using cross-validation. You can use techniques like **GridSearchCV** or **RandomizedSearchCV** to find the optimal k.