Seoul Bike Sharing Demand Prediction Project

Name: Durga Naga Padmasree Sappa

Course: DS 861

Prof: Eghbal Rashidi

Table of Contents

Seoul Bike Sharing Demand Prediction Project

Dataset Link

Video Link

Introduction

With cities growing smarter and greener, bike-sharing has become a key part of sustainable urban transport. It's affordable, convenient, and eco-friendly — and in Seoul, it's already a big part of how people get around.

But as more people rely on these bikes, a new challenge shows up: demand can be unpredictable. Some days, bikes are everywhere; other days, the docks are empty. That's what led me to ask this:

Can we predict bike rental demand accurately, using just weather and calendar data?

If we can, it opens up a lot of opportunities: better planning, fewer empty docks, smarter staff scheduling, and even targeted promotions. So I decided to dive in and see what the data could tell me.

Project Objective

My main goal here was to build a machine learning model that could predict how many bikes would be rented at a given hour in Seoul.

But honestly, this wasn't just about the model. I wanted to understand **why** people rent bikes — what patterns show up with time, temperature, or holidays? And can we turn those patterns into something useful?

So through this project, I set out to:

- Explore how factors like temperature, holidays, and seasons impact rental behavior
- Build models that are not just accurate, but also make sense
- Pull out insights that could actually help improve how the system runs in real life

I structured the whole process using the CRISP-DM methodology — from understanding the business problem to exploring the data, building models, tuning them, and finally evaluating the results.

This notebook will walk through that journey.

Phase 1: Data Understanding

STEP 0: Import Libraries

Before jumping into the data, I started by setting up my environment — importing all the libraries I'd need for the rest of the project. This step might seem small, but it's like prepping your tools before working on something intricate. A good setup makes the whole process smoother.

Here's what I brought in:

- pandas and numpy — my go-to libraries for handling data and doing numerical operations.
- matplotlib.pyplot and seaborn — for visualizing trends, distributions, and relationships in the data.
- From sklearn, I pulled in several modules:
    - model_selection for splitting the data and running cross-validation.
    - Preprocessing for scaling and encoding features.
    - Metrics to evaluate model performance.
    - And a range of **regression and classification models**, since I wanted to test different approaches and compare how they perform.

This mix gives me all the flexibility I need — from cleaning and preparing the data to building, tuning, and evaluating models.

Ignore warnings

I decided to suppress some of the non-critical warnings. During model development, especially when tuning or running cross-validation, scikit-learn can throw a lot of messages. While some are useful, many just clutter the output, so I muted the ones I already understood to keep things tidy and focused.

STEP 1: Load the Dataset

To kick things off, I loaded the Seoul Bike Sharing dataset from my local drive using pandas.read_excel. This dataset contains **hourly bike rental records**, along with a mix of **weather**, **calendar**, and **system status** features.

At this stage, my goal was simple: make sure the data loads correctly and get a quick look at its structure.

I started by checking:

- The **shape** of the dataset — to see how much data I'm working with.
- The **first few rows** — to get a feel for the column names, data types, and the overall layout.

Result Interpretation

The dataset loaded successfully without any errors.

It contains 8,760 rows and 14 columns, which makes sense: there are 365 days × 24 hours = 8,760 hourly records in a year. So we're dealing with complete hourly coverage.

The features look quite rich. Each row includes:

- A date, hour, and bike rental count.
- Weather-related data like temperature, humidity, wind speed, visibility, solar radiation, rainfall, and snowfall.
- Calendar-based fields like season, whether it's a holiday, and if the system was functioning.

This gives us everything we need to start exploring how external factors influence rental demand — and eventually build a predictive model around it.

STEP 2: Initial Data Exploration

Before jumping into preprocessing, I wanted to take a step back and understand the raw structure of the dataset. At this point, my goal wasn't to clean or engineer features yet — just to get a solid feel for what I'm working with.

Here's what I checked:

- **Data types**: I needed to make sure every column was in the expected format. For example, the Date column should be a datetime object, and weather features should be numeric.
- **Missing values**: Missing data can seriously affect model performance, so I wanted to catch that early.
- **Descriptive statistics**: By looking at means, min/max values, and standard deviations, I could start spotting possible outliers or weird entries.
- **Duplicate rows**: Sometimes datasets have repeated entries that can skew analysis or model training, so I checked that too.

Result Interpretation

- The data types looked great. Most of the features were already in the correct format, including the Date column, which was already parsed as a datetime. That'll be really helpful when I start extracting time-based features.
- There were **no missing values** at all, which is always a relief. That saves time and avoids the risk of introducing bias through imputation.
- The descriptive stats told me a lot:
    - The **target variable**, Rented Bike Count, had a wide range, from 0 to 3,556, with a mean around 704. That kind of variability means I'll need models that can handle skewed data and potential outliers.

    - Features like **Temperature**, **Humidity**, and **Visibility** showed large variation, which aligns with Seoul's seasonal climate and weather patterns.
    - For **Rainfall**, **Snowfall**, and **Solar Radiation**, most of the values were zero, but some spikes were clearly visible, indicating

STEP 3: Data Cleaning, Preprocessing & Feature Engineering

Once I had a good grasp of the raw data, the next step was to clean it up and make it more usable for modelling. This involved a mix of renaming, reformatting, and creating new features that could help the model pick up on important patterns.

What I Did

- **Renamed columns** to replace spaces with underscores. This makes the code cleaner and avoids syntax errors later.
- **Converted the Date column** into a proper datetime format. This is important because it allows me to extract useful time-based features.
- From that Date, I pulled out:
    - **Year**, **Month**, and **Day**
    - **Day of the week** (0 = Monday, 6 = Sunday)
    - A new feature called **Is_Weekend**, which flags Saturdays and Sundays
- Then, I handled the **categorical variables**:
    - I used LabelEncoder to convert text columns like Holiday, Functioning Day, and Seasons into numeric values that machine learning models can understand.
- Lastly, I **dropped the original Date column**, since I had already extracted everything I needed from it.

Result Interpretation

After this step:

- Every single column in the dataset is now **numeric**, which is perfect for feeding into most machine learning models.
- The newly created features — especially Day_of_Week and Is_Weekend — will help capture **temporal usage patterns**. For example, rentals on weekends might behave very differently compared to weekdays.
- Encoding the text features ensures there's **no data leakage** and keeps things consistent across training and evaluation.
- At this point, the dataset is much more structured and **ready for scaling and model building**.

This was a crucial step in transforming raw, human-readable information into something a model can actually learn from.

STEP 4: Outlier Detection

Before diving into modeling, I wanted to take a closer look at the numerical features — not just through stats, but visually. Outliers can quietly throw off model performance by skewing the scale of features or pulling the model's attention toward rare, extreme events.

So, I started with **boxplots** to visually inspect the spread and skew of key numeric columns.

What I Looked At

I focused on 8 continuous features that are likely to impact rental behavior:

- **Temperature**, **Humidity**, **Wind Speed**, **Visibility**
- **Dew Point Temperature**, **Solar Radiation**
- **Rainfall** and **Snowfall**

These are all environment-driven variables that can fluctuate drastically, which also means they can contain anomalies or extreme cases.

Results Interpretation

The boxplots immediately showed some interesting patterns:

- **Temperature, Humidity, Dew Point, and Visibility** had fairly normal distributions with just a few mild outliers — nothing too worrying.
- **Wind Speed** had some higher spikes that stood out.
- **Solar Radiation** was heavily skewed — most values were near zero, but a few were significantly higher.
- **Rainfall and Snowfall** were mostly zero, with a few big jumps — classic outlier behavior.
- **Correlation Heatmap**: Helps understand linear relationships. For example, **Temperature** and **Dew Point** are strongly related, while **Rainfall** isn't strongly tied to much at all.
- **Distribution Plot of `Rented_Bike_Count`**: Still right-skewed, with most values under 1,000. That suggests I might need models that can handle skewed targets or consider log transformation later.

Outlier Removal Using the IQR Method

To clean this up, I used the **Interquartile Range (IQR)** method to filter out extreme values. It's a simple but effective way to trim the extremes without touching the core patterns.

For each feature:

- I calculated Q1 (25th percentile), Q3 (75th percentile), and the IQR.

- Any data point falling outside Q1 - 1.5*IQR or Q3 + 1.5*IQR was removed.

Result Interpretation

- **Initial shape**: (8760, 18)
- **Final shape after removing outliers**: (7006, 18)

That means I removed **1,754 records**, which is around **20% of the dataset**. It sounds like a lot, but considering these were mostly extreme weather conditions (like snowstorms or high solar radiation), it makes sense to remove them. These points are rare and not reflective of typical bike usage behavior.

Decision - Not Removing Outliers

However, I realized that this outlier removal process was also eliminating all instances of rainfall and snowfall, which are critical to understanding weather-related bike usage behavior. Given the significance of this 20% of the data, I have decided to proceed with the original dataset, including outliers, to preserve the integrity and completeness of the analysis.

STEP 5: Exploratory Data Analysis (EDA)

Before jumping into model training, I wanted to take a step back and explore the data visually. My goal here was simple: understand how bike rental demand varies across time, seasons, and special conditions like holidays and weekends.

These insights help in two ways:

1. They **validate** whether the features I engineered are meaningful.
2. They **inform** how the final model should be structured, especially which interactions or seasonal effects to account for.

What I Explored

I created a series of **box plots** and a **line plot** to answer some key questions:

- What does the overall rental distribution look like?
- How does demand shift between weekdays and weekends?
- Do holidays significantly change usage patterns?
- Which seasons see the most activity?
- What times of day are bikes used the most?

Results Interpretations

1. **Overall Rental Count Distribution**

Most rental counts fall between **250 and 1000**. There’s a long tail and a few big outliers, likely from high-usage days like sunny weekends or public events.

2. **Rental Count by Season**

Seasonality plays a clear role:

- **Summer (Season 2)** has the highest median demand.
- **Winter (Season 3)** shows a sharp drop in rentals. This tells me that weather-related features (like temperature) will be important for modeling.

3. **Rental Count by Hour**

There's a distinct **bimodal peak** at **8 AM and 6 PM**, matching common commute times.

- Usage is almost zero overnight (2–5 AM). This pattern makes "Hour" one of the strongest predictors of demand.

4. **Rental Count on Holidays**

As expected, **non-holidays show higher usage**.

- Holidays have lower, more consistent rentals — people may stay home or use other transport.

5. **Rental Count by Day of Week**

Weekdays (Monday–Friday) are fairly consistent.

- Weekends drop a bit in median rentals but have higher variability — probably due to leisure usage. This supports using a **weekday/weekend flag** rather than separate models per day.

6. **Weekday vs Weekend Rental Count**

Weekday rentals are **higher and more spread out**, showing peaks due to structured workday routines. Weekend usage is **lower and more stable**, which could guide bike availability and maintenance planning.

7. **Line Plot: Hourly Patterns (Weekday vs Weekend)**

This was one of the most revealing plots.

- **Weekday rentals spike sharply at 8 AM and 6 PM**, showing classic commuting patterns.
- **Weekend rentals are flatter**, with a smoother rise and fall during the afternoon.

This distinction is crucial — it means my model needs to account for different **hourly behaviors** depending on whether it's a weekday or weekend.

Key Takeaways (for Modeling & the Business)

- Time-related features like **hour, day of week, and weekend indicator** are clearly important.
- Weather and seasonal data will help explain lower demand in colder months.
- Holidays, though less frequent, shift demand patterns enough to matter.

- These insights don't just shape the model — they also help businesses **optimize fleet usage**, **schedule maintenance**, and **target promotions** effectively.

Phase 2: Data Preparation and Modeling

With the data now cleaned and explored, it was time to prepare it for machine learning. This phase involved defining what I wanted the model to predict, scaling the data properly, splitting it for training and testing, and creating some baseline models to benchmark performance against.

STEP 6: Feature-Target Split

To start, I needed to define two essential components for supervised learning:

- X → the input features (everything the model will learn from)

- y → the target variable (what the model is trying to predict)

In this case, the target is **Rented_Bike_Count** — the number of bikes rented at a specific hour. Everything else — from temperature and humidity to hour of the day and whether it's a weekend — acts as a predictor.

To avoid any data leakage or unnecessary noise, I also dropped the two string-based columns I had created purely for visualization: Day_Name and Weekend_Label.

Result Interpretation

After splitting:

- X contains **17 features** — these include numeric weather attributes, time-based variables, and encoded categorical fields.

- y is a one-dimensional series containing the actual bike rental counts for each row.

This separation gives me the structure I need to move forward with scaling and model training. From this point on, all machine learning steps will treat X as the input and y as the outcome we're trying to predict.

STEP 7: Train-Test Split & Feature Scaling

With my features ready, the next step was to split the dataset into training, validation, and testing sets.

This is a core part of any supervised learning workflow — the idea is to train the model on one portion of the data (the training set), tune it on another (validation set), and evaluate its performance on a completely separate portion (the test set) that the model hasn't seen before. This way, I can get a realistic sense of how well the model will perform on new, unseen data.

I went with a standard **80-20** split at first, and then further split the training portion:

- **64%** for **training** — this is where the model will learn patterns and relationships
- **16%** for **validation** — used for tuning and model selection
- **20%** for **testing** — this set is held back to check how well the model generalizes

To ensure reproducibility, I used a fixed random_state.

After splitting, I needed to normalize the feature values. This step is crucial, especially for algorithms that are sensitive to the scale of input data.

Some models, like K-Nearest Neighbors (KNN), Support Vector Machines (SVM), and Neural Networks (MLP), rely on distance calculations or gradient descent. If one feature (like visibility, which ranges up to 2000) has much larger values than others (like wind speed), it could end up dominating the learning process unfairly.

To fix this, I applied **standard scaling** using StandardScaler, which:

- **Centers** each feature around 0
- **Scales** features to have unit variance (standard deviation = 1)

This helps ensure that all features are treated equally during training.

Result Interpretation

- The training set contains **5,606** records
- The validation set contains **1,402** records
- The test set contains **1,752** records

This confirms that the split worked correctly. From this point on:

- Only the **training set** will be used for learning
- The **validation set** will be used for model selection and tuning
- The **test set** stays completely untouched until the final evaluation

This ensures the results are realistic and not biased or overfit to data the model has already seen.

After applying scaling:

Every feature in the scaled datasets now has a mean of approximately 0 and a standard deviation of 1.

This transformation doesn’t affect the model’s ability to learn — it just ensures a fair starting point across all features.

Standardizing the data like this helps with:

- Faster convergence during training
- More accurate predictions, especially for models that are **scale-sensitive** (like KNN, SVM, and MLP)

With the scaled features ready, I was set to move into the next phase: **building baseline models and comparing their performance.**

STEP 8: Data Leakage Check

Before finalizing the model, I checked for **data leakage**, where input features might unintentionally reveal or overlap with the target (Rented_Bike_Count). This is crucial to ensure the model is learning genuine patterns rather than memorizing or exploiting hidden clues.

- I calculated the **correlation of each feature with the target**.
- Looked for any variables with **suspiciously high correlation** (typically > 0.9), which could indicate leakage.
- Visualized correlations using a heatmap for clarity.

Result Interpretation

- **No evidence of data leakage**:
  - The highest correlation was with Temperature(°C) at **0.54**, which is reasonable and expected.
  - Other top correlations like Hour, Dew Point Temperature, and Solar Radiation are all **logically related** to bike rental behavior.
- **No features** had near-perfect correlation with the target.
- Rainfall(mm) and Snowfall(cm) returned NaN — likely due to having **zero variance** (all or mostly zero values).

This confirms that the modeling process is trustworthy and the evaluation results are reliable.

STEP 9: Dummy Models

Before training any real machine learning models, I wanted to establish a **baseline** using a DummyRegressor. This step might seem simple, but it's really important — it tells me whether the models I build are actually learning anything meaningful or just overcomplicating the problem.

The dummy model works like this:

- It **ignores all input features**
- It simply **predicts the mean rental count** from the training set, no matter the input

This gives me a baseline level of performance. If my future models don't significantly beat this, it means something's wrong — maybe the features aren't useful, or the model is overfitting or underfitting.

What I Did

I trained the dummy model on the training data (y_train) and predicted on the test set (X_test). Then, I evaluated its performance using three standard regression metrics:

- **R^2 Score** (how much variance is explained)
- **RMSE** (Root Mean Squared Error)
- **MAE** (Mean Absolute Error)

Result Interpretation

- **R^2 Score**: ~0
  This is expected — the dummy model doesn't explain any variation in rental demand. It's just predicting the average over and over.
- **RMSE**: ~635
- **MAE**: ~512
  These values show the average prediction errors. They give me a concrete target to beat — any real model needs to produce lower error metrics than this.

This baseline helps validate that everything is wired up correctly. Now that I've set a minimum benchmark, I can move forward and start training real regression models — and see how much better they can do.

STEP 10: Model Training (Regression)

With the data fully prepared, it was finally time to train some real models.

Rather than jumping straight to a favorite algorithm, I decided to take a **broad approach** — training a variety of regression models to see which one works best for predicting bike rental demand. Each model type brings a different learning style to the table, from simple linear fits to complex ensemble strategies.

Models I Trained

I grouped the models into a few categories:

- **Linear Models**:
  - Linear Regression
  - Ridge
  - Lasso
  - ElasticNet
- **Tree-Based Models**:
  - Decision Tree
  - Random Forest
  - Gradient Boosting

- **Others**:
    - K-Nearest Neighbors (KNN)
    - Support Vector Regressor (SVR)
    - Multi-Layer Perceptron (MLP) — a basic neural net

Each model was trained on the training set and evaluated on the test set using three key metrics:

- **R² Score** — how much variance in rental count the model explains
- **RMSE** — penalizes larger errors more harshly
- **MAE** — gives the average size of errors, regardless of direction

Result Interpretation

- **Gradient Boosting** clearly outperformed all other models across every metric. It captured the complexity in the data well and handled the skewed rental distribution effectively.
- **Random Forest** and **Decision Tree** models also did quite well, which supports the idea that tree-based models work well on this kind of structured, non-linear data.
- **MLP (Neural Network)** did reasonably well but didn't quite match the ensemble models — possibly due to hyperparameter limitations or needing more data.
- **Linear models** (including Ridge, Lasso, ElasticNet) didn't perform well at all, with R² stuck around 0.54. This suggests that the relationship between features and rental demand is **not linear**.
- **SVR** performed the worst. It likely struggled due to scaling sensitivity or the lack of kernel tuning.

Final Thought

This comparison made it clear that **ensemble models — especially Gradient Boosting — are the best fit for this prediction task**. They're flexible, handle non-linear patterns well, and are generally robust across many real-world scenarios like this one.

STEP 11: Visualize Model Comparison

After training and evaluating all the regression models, I wanted to visualize their performance across the three key metrics:

- **R² Score** → higher is better (measures explained variance)
- **RMSE** → lower is better (penalizes large errors)
- **MAE** → lower is better (shows average prediction error)

These bar plots make it easier to compare models side by side and spot performance trends that aren't always obvious in a table.

Results Interpretation

1. **R² Score**

- **Gradient Boosting** leads the pack, with an R² close to **0.92**, meaning it explains almost all the variation in bike rental counts.
- **Random Forest** and **Decision Tree** models also perform well, followed by the **MLP Regressor**.
- Linear models (including Ridge, Lasso, ElasticNet) and **SVR** fall significantly behind, confirming that this problem is **non-linear** in nature.

2. **RMSE**

- Again, **Gradient Boosting** comes out on top with the **lowest RMSE (~173)**, indicating fewer large prediction errors.
- The tree-based models show consistent and reliable performance.
- SVR’s RMSE is the highest by far, reinforcing its poor fit for this dataset.

3. **MAE**

- Gradient Boosting also posts the **lowest MAE (~109)**, suggesting it doesn’t just do well on average — it also tends to be close to the actual values on a consistent basis.
- Random Forest and Decision Tree follow behind with competitive results.
- Linear models and SVR show larger average errors, again emphasizing their limitations here.

Conclusion for This Phase

- **Gradient Boosting** is clearly the best-performing model across all evaluation metrics. It will be the focus for further refinement through hyperparameter tuning and cross-validation.
- **Tree-based models in general** are very effective for this problem — they’re able to capture non-linear patterns and interactions in the data.
- **Linear models** don’t offer the flexibility needed here, and **SVR** failed to generalize well, likely due to scale sensitivity and lack of kernel tuning.

This visualization confirmed that the path forward is clear: **focus on Gradient Boosting**, tune it properly, and interpret what it's learning using feature importance tools.

Phase 3: Cross-Validation and Hyperparameter Tuning

STEP 12: Cross-Validation (k-Fold)

After seeing how the models performed on a single train-test split, I wanted to take things a step further and validate their stability. Sometimes, a model can look great on one split but perform poorly on another. To avoid this trap, I used **5-fold cross-validation**.

Why Cross-Validation?

Cross-validation helps measure how consistent a model is across different subsets of data. Here's how it works:

- The training data is split into 5 equal parts (folds)
- The model is trained on 4 folds and validated on the remaining one
- This process repeats 5 times, so each fold gets used for validation exactly once

The result is a **mean R^2 score** and **standard deviation**, giving both performance and reliability.

I reused all previously defined models and ran cross_val_score() with:

- cv = 5 (5 folds)
- scoring = 'r2' (since R^2 is my primary evaluation metric)

This gave me a clearer, more reliable estimate of how each model generalizes.

Result Interpretation

The bar chart clearly highlights:

- **Gradient Boosting** and **Random Forest** as the top performers — both accurate and consistent.
- **MLP** and **Decision Tree** also do well, but with slightly higher variability.
- **Linear models** cluster around the same mediocre performance, consistent, but not strong enough.
- **SVR** once again struggles, with both low accuracy and limited stability.

What This Means

Models with **high mean R^2 and low variance** are generally safer bets when moving toward production. Based on this:

- **Gradient Boosting** is the strongest candidate for fine-tuning.
- Random Forest also deserves attention.
- Others like MLP or Decision Tree may still be useful with additional tuning, though they aren't as stable.

This gives me confidence in which models to prioritize for the next phase: **hyperparameter tuning** and **deep dives into model interpretability**.

STEP 13: Hyperparameter Tuning

Even the best models can fall short if their settings aren't optimized. To push the top-performing models further, I used **GridSearchCV** to systematically tune their key hyperparameters — the ones that control how the models learn and generalize.

This step is all about finding the sweet spot:

- Not too simple (underfitting)
- Not too complex (overfitting)
- Just right for the patterns in our data

Models Tuned & Parameters Optimized

I focused on **six models** that had shown strong or promising performance in earlier evaluations:

1. **Random Forest**
   - Parameters tuned: n_estimators, max_depth, min_samples_split
2. **Gradient Boosting**
   - Parameters tuned: n_estimators, learning_rate, max_depth
3. **Ridge Regression**
   - Parameter tuned: alpha
4. **KNN**
   - Parameter tuned: n_neighbors
5. **Decision Tree**
   - Parameters tuned: max_depth, min_samples_split
6. **MLP Regressor (Neural Net)**
   - Parameters tuned: hidden_layer_sizes, max_iter

Result Interpretation

- **Gradient Boosting** delivered the best performance (**R^2 = 0.925**), confirming it's both accurate and highly tunable with the right depth and learning rate.
- **Random Forest** followed closely (**R^2 = 0.901**), benefiting from more estimators and deeper trees.
- **Decision Tree** improved with tuning (**R^2 = 0.838**) but still falls short of ensemble methods.
- **MLP Regressor** reached **R^2 = 0.830**, showing promise but requiring more computation and careful tuning.

- **KNN** slightly improved (**R^2 = 0.764**) with 7 neighbors but lacks robustness compared to other models.
- **Ridge Regression** (and other linear models) saw little to no gain even after tuning (**$R^2 \approx 0.556$**), reinforcing their limitations for this non-linear problem.

Takeaway

These results reinforced that **Gradient Boosting is the best choice** going forward, not only because it performs well, but also because it responds well to tuning. It balances accuracy, consistency, and control.

With hyperparameters now optimized, the next step is to analyze how well the model generalizes using **learning curves** and dig deeper into **bias-variance tradeoffs**.

STEP 14: Bias-Variance Analysis (Learning Curves)

To evaluate how well the top models generalize, I plotted **learning curves** and ran a **bias-variance diagnostic**. This step helps identify whether a model is underfitting (high bias), overfitting (high variance), or generalizing well.

Why Learning Curves?

Learning curves show how model performance changes as the training set size increases. They help answer key questions:

- Is the model learning enough from the data?
- Would more data help?
- Is the model too simple or too complex?

Result Interpretation

- **Gradient Boosting & Random Forest** showed textbook behavior: training scores near 1.0, and validation scores that steadily improve and plateau at high values (~0.95). These are robust, low-bias, low-variance models.
- **Ridge Regression** had a flat curve — it doesn't benefit from more data and clearly underfits the problem.
- **MLP and Decision Tree** showed good fits, but are more sensitive to parameters and data size.
- **KNN** improved with more data but topped out earlier than the ensembles.

Takeaways

- **Gradient Boosting and Random Forest** are the most balanced and generalizable models.
- **Ridge Regression** is too simple for this task, confirming earlier results.
- Learning curves helped visually confirm model stability and provided confidence in choosing the final model for deployment.

Next step: dig into **feature importance** to understand what's driving predictions.

STEP 15: Residuals and Predictive Error Analysis

To further validate the quality of predictions, I analyzed the **residuals** — the difference between the actual and predicted bike rental counts — for the top 6 tuned models. This step helps check whether a model is introducing **systematic bias** or making **unpredictable errors**.

For each model, I plotted:

- **Residuals vs. Predicted values** — to detect patterns or non-random behavior
- **Residual distribution histograms** — to check whether errors are symmetric and centered around zero

Result Interpretation

- **Gradient Boosting & Random Forest**: Residuals are tight, symmetric, and centered around zero, indicating accurate, unbiased, and stable predictions.
- **Decision Tree & MLP Regressor**: Slightly more spread, but residuals remain well-distributed with no major bias — overall good generalization.
- **KNN Regressor**: Wider residual spread with underprediction at higher values — moderately reliable.
- **Ridge Regression**: Residuals are scattered and skewed — clear signs of underfitting and poor model fit.

Conclusion: Ensemble models (especially Gradient Boosting) produce the cleanest residuals, validating their effectiveness for this regression task.

STEP 16: Feature Importance Analysis

After selecting the best-performing models — particularly **Gradient Boosting**, **Random Forest**, and **Decision Tree** — I examined which features contributed most to their predictions.

Tree-based models naturally calculate **feature importance** based on how much each variable reduces prediction error.

Result Interpretation

- **Hour of the day** was by far the most important feature across all models, highlighting the impact of commute times and daily routines.
- **Temperature** was consistently the second-most important predictor, as weather directly affects outdoor activity.
- **Functioning Day** and **Solar Radiation** also had a notable influence.
- Features like **Rainfall**, **Snowfall**, and **Wind Speed** contributed very little and could be dropped for simplified models.

Takeaway

- **Temporal and weather-related features** are the strongest indicators of rental demand.
- This analysis improves model **explainability** and helps prioritize features for future work.
- For real-time systems or lightweight deployment, **less important features can be safely excluded**.

Gradient Boosting continues to show strong performance and interpretability, making it the leading candidate for production use.

STEP 17: Data Leakage Check

Before finalizing the model, I checked for **data leakage**, where input features might unintentionally reveal or overlap with the target (Rented_Bike_Count). This is crucial to ensure the model is learning genuine patterns rather than memorizing or exploiting hidden clues.

- I calculated the **correlation of each feature with the target**.
- Looked for any variables with **suspiciously high correlation** (typically > 0.9), which could indicate leakage.
- Visualized correlations using a heatmap for clarity.

Result Interpretation

- **No evidence of data leakage**:
  - The highest correlation was with Temperature(°C) at **0.56**, which is reasonable and expected.
  - Other top correlations like Hour, Dew Point Temperature, and Solar Radiation are all **logically related** to bike rental behavior.
- **No features** had near-perfect correlation with the target.
- Rainfall(mm) and Snowfall(cm) returned NaN — likely due to having **zero variance** (all or mostly zero values).

This confirms that the modeling process is trustworthy and the evaluation results are reliable.

STEP 18: Final Model Evaluation

After full preprocessing and tuning, I evaluated the top 6 models using a standardized pipeline and tested them on unseen data. Metrics considered:

- **R² Score**: Higher is better (explained variance)
- **RMSE**: Lower is better (penalizes large errors)
- **MAE**: Lower is better (average absolute error)

Result Interpretation

- **Gradient Boosting** once again leads, offering the best balance of low error and high accuracy. Its R^2 of **0.92** confirms it's both powerful and generalizes well.
- **Random Forest** is a close second, and also highly stable.
- **Decision Tree** and **MLP** show decent performance but fall short of ensemble models.
- **KNN** and **Ridge** lag behind, especially Ridge, which underfits despite tuning.

Conclusion

When I started this project, I had one central question in mind:

Can we predict how many bikes will be rented in Seoul at a given hour using just weather and calendar data?

It sounded simple — almost too simple. But I quickly realized this wasn't just a technical challenge. It was about translating patterns in human behavior into something quantifiable, actionable, and useful for real-world planning.

The Journey: From Data to Decision-Making

I began with raw hourly rental data from Seoul's bike-sharing system, coupled with environmental and calendar information. After cleaning and exploring the data, I didn't just see numbers — I saw routines.

- People tend to ride less frequently when it's cold or wet.
- Rentals spike like clockwork at 8 AM and 6 PM.
- Weekends look different from weekdays.
- Holidays flatten demand altogether.

These weren't just trends — they were **signatures of how people live and move through the city**.

From there, I tried different modeling approaches. I could've gone with classification and bucketed demand into ranges. But that wasn't the kind of answer I was looking for. I didn't want to say "expect high demand." I wanted to say:

"Expect 1086 rentals tomorrow at 8 AM if it's 21°C and sunny."

That's why I chose regression. It gave me the resolution I needed to honor the real-world nature of the problem.

What I Built

After training and comparing several models — from linear regression to neural networks — one model stood out: **Gradient Boosting Regressor**.

It wasn't just the most accurate (R^2 = **0.92** on the test set) — it was also the most stable, generalizable, and interpretable.

I validated it through:

- **Cross-validation** (low variance across folds)
- **Learning curves** (clear convergence)
- **Residual plots** (no bias or systematic error)
- **Feature importance** (results that matched human intuition)

Then I tested it with a real-world input:

8 AM on a weekday in July, with 21°C, clear skies, and normal conditions.
Predicted demand: 1086 bikes.

What the Data Revealed

In the end, this wasn't just a machine learning project. It became a data-driven story about urban behavior:

- **Routine drives demand.** We ride when we commute.
- **Comfort matters.** Weather changes how we move.
- **Behavior is non-linear.** Tree-based models captured that better than anything else.
- **Time and context shape usage.** Holidays and seasons matter more than we think.

Why It Matters

A city like Seoul doesn't just need forecasts — it needs **insightful, operationally useful intelligence**.

This model can help with:

- **Fleet rebalancing**
- **Staff scheduling**
- **Predictive maintenance**
- **Targeted promotions**

It can turn daily planning into a **proactive**, data-informed process — one that keeps the city running smoothly and sustainably.

Final Reflection

Bike rentals aren't random. They're shaped by time, temperature, and context.

And now, thanks to this project, **they're predictable.**

Challenges & Lessons Learned: My Data Mining Journey

This project wasn't just about building models — it was about learning how to think, plan, and evaluate like a data scientist. I made mistakes, asked questions, and constantly refined my workflow as I understood the "why" behind every step. Each challenge helped me get better, not just at coding, but at thinking critically about data.

- At the start, I did what most beginners do — I split the dataset into training and testing sets and thought I was done. But I quickly realized that I wasn't validating anything during model tuning. I learned that I needed a proper train-validation-test split to avoid overfitting and to make fair comparisons between models. Switching to a 64-16-20 split made a big difference and gave me a better grip on model selection.

- I also made the classic mistake of scaling all my features before splitting the data. I didn't realize I was leaking information from the test set into my training process. Once I understood this, I corrected the flow, first splitting the data, then applying StandardScaler only to the training set, and using it to transform the validation and test sets. It felt like a small change, but it was a huge lesson in maintaining clean pipelines and avoiding subtle leakage.

- Initially, I skipped using a dummy model. I was excited to try fancy regressors and see big R^2 scores. But I added a Dummy Regressor later to establish a baseline, and it changed how I viewed my entire evaluation process. The dummy taught me what "not learning anything" looked like. It gave me a concrete benchmark — if my models didn't beat the dummy, they weren't really doing anything useful.

- Another moment of realization came during cross-validation. I had been using unscaled data in cross-validation, which made models like KNN and MLP perform terribly. Once I started using scaled features (X_train_scaled), the performance and reliability of my cross-validation results improved dramatically. It reminded me how preprocessing must always align with model assumptions.

- Hyperparameter tuning turned out to be more important than I expected. At first, I used all models with default settings, but then I set up GridSearchCV for models like Random Forest, Gradient Boosting, Ridge, and MLP. The improvement, especially for Gradient Boosting, was noticeable. I also learned how to design manageable grids to avoid wasting time and computational power.

- While comparing model performance, I realized I didn't fully understand why some models failed. So I started plotting learning curves and looking at the gap between training and test scores. That helped me figure out which models were overfitting, underfitting, or generalizing well. I finally understood how bias and variance actually work, not just as theory, but as patterns I could see in the data.

- Metrics like RMSE and MAE were useful, but they didn't tell the whole story. When I began plotting residuals — looking at residuals vs. predicted values and their distributions — I saw a new side of model evaluation. A model could have a decent score, but if the residuals were skewed or patterned, it meant the model wasn't truly balanced. These visual checks gave me more confidence in how my models were behaving.

- I also got into the habit of checking feature importance. Before, I just assumed all features helped equally. But when I used. feature_importances_ from tree-based models, I saw that features like Hour, Temperature, and Dew Point were the real drivers of prediction. It was validating to see that this lined up with what I had observed during EDA. It helped me justify my modeling decisions and trust my interpretations.

- One mistake I made early on was accidentally using the test set during tuning. It didn't seem like a big deal until I realized the test set should only be used at the very end. From that point on, I reserved the test set for final evaluation (Step 18 only) and relied solely on the validation set for all comparisons and tuning.

- The moment everything came together was when I used my best model (Gradient Boosting) to make a real-time prediction. I input a realistic scenario — Monday morning in July — and got an expected rental count. That simple action felt like a huge win. It wasn't just about metrics anymore — it was about applying the model like it would be used in the real world.

This project was full of technical lessons, but more importantly, it taught me how to structure my thinking, how to evaluate models honestly, and how to approach problems with both logic and curiosity. I walked in wanting to build a model. I walked out understanding the entire process — and knowing how to make it count.