Reading the Dataset

- **Use Pandas to load data**

```
df = pd.read_csv("data.csv")
print(df.head())        # Preview first 5 rows
```

- **Quick checks:** Verify the shape and basic info. Use `df.shape` to get number of rows/columns and `df.columns` for column names.

```
print("Shape:", df.shape)      # (rows, columns)
print("Columns:", df.columns)
```

- **Data types and memory:** Use `df.dtypes` or `df.info()`. This shows each column's data type (int, float, object, etc.) and non-null counts, helping identify numeric vs categorical features and any memory heavy columns.

```
print(df.info())   # concise summary (types and non-null counts)

print(df.dtypes)

df.describe() # provides basic stats for numeric columns
```

- **Missing values:** Check how many nulls in each column. Use `df.isnull().sum()` to get a count of missing values per column. You can also see non-null counts via `df.info()`

```
print(df.isnull().sum())
```

Look for columns with a high percentage of missing data that may need special treatment. (For a quick percentage: `df.isnull().mean()*100` gives % of missing in each column.)

- **Unique values:** For categorical columns, get unique counts with `df.nunique()` or value counts with `df['col'].value_counts()`. This helps spot if a numeric column is actually categorical (few unique values) and whether classes are imbalanced.

- **Check target variable**
  `df['Y'].value_counts()`
  - If categorical target - check Imbalance

Data Exploration

Correlation (Multicollinearity)

Checking Multicollinearity (VIF)
Multicollinearity means two or more features are highly correlated, which can cause instability in linear model coefficients (inflating standard errors, making it hard to interpret coefficients). It's mainly a concern for linear regression (and other linear models) – not so much for predictive accuracy (regularization can handle it, and tree models don't mind), but for interpretability and assumptions.

```
import numpy as np
import seaborn as sns
matrix = X_num.corr() # corr = df.corr(numeric_only=True)

plt.figure(figsize=(12, 8))

sns.heatmap(matrix, annot=True, cmap='Blues')
```

- Look for pairs with very high correlation (e.g. |corr| > 0.8). If you find any, consider removing one of the two, or combining them. For example, features like Height and Weight might both correlate with an outcome; or YearBuilt and Age are perfectly correlated (Age = currentYear - YearBuilt) – one is redundant.

```
cor_pairs = matrix.unstack()

sorted_pairs = cor_pairs.sort_values(kind = 'quicksort')

strong_pairs = sorted_pairs[abs(sorted_pairs) > 0.7]

print(strong_pairs)
```

Variance Inflation Factor (VIF): A more formal measure. VIF for a feature = 1/(1 - R^2) where R^2 is obtained by regressing that feature on all other features. A VIF of 1 means no correlation with others; VIF > 5 (or >10 by some rules) indicates high multicollinearit
To calculate VIF:

```
from statsmodels.stats.outliers_influence import variance_inflation_factor
X = df[['feat1','feat2','feat3']].values  # features matrix
vif = [variance_inflation_factor(X, i) for i in range(X.shape[1])]
print(vif)  # VIF for each feature in order
```

- Or manually: for each feature, run a linear regression against all others and get R^2. But using variance_inflation_factor from statsmodels is quicker.

Handling multicollinearity:

- If two features are very correlated (say r > 0.9), you can drop one without losing much information. Alternatively, combine them (e.g., average them, if that makes sense).
- You can also use techniques like **PCA** (principal components) to reduce correlated features into a smaller set of uncorrelated components.
- For modeling, using **Ridge regression** (L2 regularization) can mitigate multicollinearity by shrinking coefficients, and **Lasso** (L1) can even drop one of the correlated features. So if using those, you may not need to explicitly remove collinear features, but it's still good to be aware of them (especially in linear regression without regularization).

Divide into numerical / categorical

```
num_data = df[["int.rate", "installment", "log.annual.inc", "dti", "fico",
"days.with.cr.line", "revol.bal"]
cat_data = df[["credit.policy", "purpose", "inq.last.6mths",
"delinq.2yrs", "not.fully.paid"]]
```

if numeric:

- check variable distribution
- if it is skew
- check extreme values/outliers

if categorical

- check distribution of the categories, if it is balanced
- check if the categorical variables are ordinal, or just categoriories without relationship
- may convert to dummy variable or numeric ordinal variable

if datetime

- convert to number of days

Numerical

Check the distribution of the numerical continuous data

```
num_data.hist(figsize = (30, 30), bins = 20, legend = False)
plt.show()
```

Develop a probability plot to find out how compacted or degress of normalisation the data is.

```
for i, col in enumerate(num_data):
    ax = plt.subplot(3, 3, i+1)
    stats.probplot(num_data[col], plot = ax)
    ax.set_title(f"Probability Plot for {col}")

plt.show()
```

Create **box plots** showing the uncertainity in the data and the outliers.

```
for i, col in enumerate(num_data):
    ax = plt.subplot(3, 3, i+1)
    sns.boxplot(y = df[col])
    ax.set_title(f"Boxplot for {col}")
plt.show()
```

Categorical

Check the distribution of the categorical data

```
cat_data.hist(figsize = (30, 30), bins = 20, legend = False)
plt.show()
```

Create box plots showing the uncertainity in the categorical data and the outliers.

```
plt.figure(figsize = (10, 10))

cat_data.boxplot()

plt.title("Box plot showing the outliers in the categorical data")

plt.show()
```

Encoding Categorical Variables

Label Encoding: Assign each unique category an integer label. For example, {"Apple":0, "Banana":1, "Cherry":2}. Use when the category is **ordinal** (has intrinsic order) or if feeding to a tree-based model (which can handle arbitrary integers for categories).

```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
df['Category_code'] = le.fit_transform(df['Category'])
cat_data["purpose"] = le.fit_transform(cat_data["purpose"].astype(str))
```

- Now a column "Category_code" will have 0,1,2,... for each category. **Be cautious:** Label encoding implies an ordinal relationship (2 > 1 > 0), which is **not true for nominal**

categories like fruit names. Only use if the algorithm is tree-based or if the categories are ordinal (e.g., "Low", "Medium", "High")

One-Hot Encoding: Convert a categorical column into multiple binary columns (dummies) – one column per category, with 1 indicating presence of that category and 0 otherwise This avoids implying any ordinal relationship.

```
df = pd.get_dummies(df, columns=['Category'], drop_first=True)
```

- This will replace the original *Category* column with new columns like `Category_Banana`, `Category_Cherry`, etc. (one fewer than total categories if `drop_first=True` to avoid redundancy). Each is 1 or 0. You can also use `OneHotEncoder` from sklearn for the same result.
  Note: One-hot encoding increases dimensionality. If a categorical feature has *many* levels, you might need to consider encoding techniques that won't blow up feature count (like Target Encoding, Frequency Encoding, or PCA on the one-hot vectors).

- **Binary/Ordinal Encoding:** If categories have a natural order, you can map them to numerical scale manually (e.g., { "Low":1, "Medium":2, "High":3 }). Ensure the order mapping makes sense.

Missing Values Handling

- **Drop missing:** `df.dropna()` can drop rows (or columns) with any NA. Use this if the dataset is large and only a few entries are missing *and* you suspect they're random.

- **Fill with a constant:** Sometimes missing can be treated as a separate category (e.g., "Unknown") for categoricals or 0/mean for numeric if that makes sense. Use `df.fillna(value)` to replace NAs with a specified value.

missing value也可能是有价值的，比如缺失employment信息是loan status有影响的;

Mean/Median Imputation: Replace missing numeric values with the mean or median of that column. This is simple and often reasonable for roughly symmetric distributions.

```
df['Col'] = df['Col'].fillna(df['Col'].median())  # fill missing with median
```

- Similarly, `df.fillna(df.mean())` fills each column's NaNs with that column's mean

Forward/Backward Fill: For time-series or ordered data, use forward-fill (`method='ffill'`) or back-fill (`'bfill'`) to propagate last seen values

```
df.sort_values('Date', inplace=True)
df['value'] = df['value'].fillna(method='ffill')  # forward-fill previous value
```

- This carries forward the last known value. It's useful if missing entries are truly carry-overs (e.g., missing sensor reading replaced by last valid reading).

- **Interpolation:** `df.interpolate()` can fill gaps with interpolated values (linear, time, etc.) – useful for sequences.

Post-imputation check: After filling, run `df.isnull().sum()` again to ensure no or expected NAs remain.

Train Test Split

Before modeling, split your data into training and testing sets to evaluate performance on unseen data. This prevents overfitting and gives an unbiased estimate of model generalization:

Using scikit-learn:

```
from sklearn.model_selection import train_test_split
X = df.drop('target', axis=1)
y = df['target']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)
OR: train, test = train_test_split(data, test_size=0.33, random_state=42)
```

- This allocates 80% data to training, 20% to testing. `random_state=42` ensures reproducibility (you get the same split every time). If classification with imbalance, use `stratify=y` to maintain class proportions in both train and test.

- **Never train on test data:** The test set simulates new, unseen data. Don't peak at it or use it for feature engineering decisions. For example, scale using only train data (as noted), and any feature selection or hyperparameter tuning should be cross-validated on the train set *only*. The test set is for final evaluation.

- **Validation set:** In practice, you may further split training into train/validation if you need to tune hyperparameters or try different models. A common approach is **train/val/test** split (e.g., 60/20/20). Alternatively, use cross-validation on the training set for tuning (see next section).

- **Shuffling:** `train_test_split` by default shuffles before splitting, which is usually desired if data has any order. (If data is time-series, you would not shuffle; instead use time-based split).

After splitting, proceed with all preprocessing (imputation, scaling, etc.) on `X_train` (fit on train, then transform train and test).

Outliers Detecting and Handling

Outliers can skew your analysis and models. (Kaggle: https://www.kaggle.com/code/nareshbhat/outlier-the-silent-killer)

Plot using box plot: `sns.boxplot(x=df['Col'])` - show outliers as points beyond whiskers (which are by default 1.5*IQR)
IQR Method: Compute the 1st quartile (Q1, 25th percentile) and 3rd quartile (Q3, 75th percentile). The IQR = Q3 – Q1. Typically, any data point outside [Q1 – 1.5IQR, Q3 + 1.5IQR] is flagged as an outlier

```
Q1 = df['Col'].quantile(0.25)
Q3 = df['Col'].quantile(0.75)
IQR = Q3 - Q1
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR
outliers = df[(df['Col'] < lower_bound) | (df['Col'] > upper_bound)]
```

- This finds all rows where 'Col' is an outlier by the IQR rule. Inspect outliers – sometimes printing a few to see if they are data errors (e.g., a height of 800 inches) or legitimate extreme values.
  Handling: If outliers are due to errors or are not relevant (e.g., an entry error like age 999), you can drop or cap them. If they are valid but skew distribution, you might apply a transformation (log, etc.) or use algorithms robust to outliers.

Z-Score Method: Calculate the z-score for each data point = (value – mean) / std. deviation. By convention, if |z| > 3 (more than 3 std dev away from mean), consider it an outlier

```
import numpy as np
col = df['Col']
z_scores = (col - col.mean())/col.std()
outliers = df[np.abs(z_scores) > 3]
```

- This flags points more than 3σ from the mean. Note **that this assumes a roughly normal distribution of the column**; if the data is highly skewed, the IQR method is usually preferred. Also, one can use scipy.stats.zscore for convenience.

```
# Detect outliers in combined data set
def detect_outlier(feature):
    outliers = []
    data = df[feature]
    mean = np.mean(data)
    std =np.std(data)

    for y in data:
        z_score= (y - mean)/std
        if np.abs(z_score) > 3:
            outliers.append(y)
    print(f"\nOutlier caps for {feature}")
    print('  --95p: {:.1f} / {} values exceed that'.format(data.quantile(.95),
                                                           len([i for i in data
                                                                if i > data.quantile(.95)])))
    print('  --3sd: {:.1f} / {} values exceed that'.format(mean + 3*(std), len(outliers)))
    print('  --99p: {:.1f} / {} values exceed that'.format(data.quantile(.99),
                                                           len([i for i in data
                                                                if i > data.quantile(.99)])))
```

```
# Determine what the upperbound should be for continuous features in dataframe.
for feat in num_data:
    detect_outlier(feat)
```

Handling Outliers

- Drop them: Simplest approach if you have enough data and outliers are clearly anomalies. Example: df = df[df['Col'] <= upper_bound] and similar for lower

bound to filter them out.

- Cap (Winsorize): Replace outliers with the boundary values (set values above upper_bound to exactly upper_bound, etc.). This retains data points but tames extreme values.

- Transform: For skewed distributions (e.g., income), a log transform can reduce the impact of large outliers by compressing the scale (see “Target Transformation” below).

- Use robust methods: Some models (e.g., tree-based) are less sensitive to outliers. Or use metrics like median absolute deviation for analysis.

```
# Capping features in df to remove outliers in numerical features
# Upper bounded outliers
for var in ['int.rate' ,'installment', 'log.annual.inc', 'fico',
'days.with.cr.line', 'revol.bal', 'not.fully.paid']:
    df[var].clip(upper=df[var].quantile(.95), inplace=True)
# Lower and Upper bounded outliers
for var in ['log.annual.inc']:
   df[var].clip(lower = df[var].quantile(.05), upper =

df[var].quantile(0.95), inplace=True)
```

Check for the presence of outliers again

```
numerical_df = df[num_data_features]
plt.figure(figsize = (15, 15))

def num_plot(df, a, var):
    ax = plt.subplot(3, 3, a+1)
    sns.boxplot(y = df[var])
    ax.set_title(f"Boxplot for {var}")
    plt.tight_layout()

for i, col in enumerate(numerical_df):
    num_plot(numerical_df, i, col)
```

Skew / Log

If the **target variable** (what you're trying to predict) is skewed or has a long tail (common with prices, incomes, counts), a transformation can improve model performance.

Plot check target variable:

```
# Plot histogram of House selling prices

sns.histplot(data['SalePrice'], bins=20, kde=True)

plt.title('House Prices Histogram')

# Probability plot

fig = plt.figure()

res = stats.probplot(data['SalePrice'], plot=plt)

plt.show()

Also Check y.skew() # apply log transformation if > 1
```

Based on the graphs above, our target variable seems to have a positively skewed distribution(long tail on the right). By plotting the data against its theoretical quantiles, we observe a **non-linear relationship** which indicates that our distribution isn't normal. The skew is also far from 0.

Log Transform: Taking $\log(y)$ (or $\log_{10}$ or natural log) can normalize a right-skewed distribution. For example, if you're predicting income, logging the target will compress high incomes and spread out lower incomes, often making the distribution more symmetric.

```
import numpy as np
y_train_log = np.log1p(y_train)  # log(1+y) to handle zero values
```

Plot again

```
sns.histplot(y_train_log, bins=20, kde=True)
```

- Here `np.log1p` is used instead of `np.log` to safely handle 0s (since $\log(0)$ is undefined; log1p(0) = 0). Remember to transform predictions back (exponentiate) to interpret in original units.
  After training a model on $\log(y)$, you'd get predicted $\log(y)$; apply `np.expm1` to get back to original scale. (expm1 is inverse of log1p).

- **When log helps:** If `y.skew()` is high (say > 1), log can reduce skewnessmedium.com. It stabilizes variance – often the case that variance of residuals grows with y, so logging can make residuals more homoscedastic. Many models (linear regression in

particular) assume residuals ~ normal; logging y can help achieve that if original y was log-normal.

- **Other transforms:** If y has zeros or negatives that log can't handle, consider **Box-Cox transform** (scipy `stats.boxcox`, which finds an optimal power transform including log as a special case). If y is left-skewed (long tail on left), you might use y^2 or similar. In practice, log (for right-skew) or sometimes sqrt for count data (since variance of Poisson ~ mean, sqrt can stabilize) are common.

- **Classification target:** If this is a classification problem (target is categorical), you wouldn't "transform" it like this; instead, you deal with class imbalance or multi-class encoding (see imbalance section).

Note: Evaluate model performance in original units as well. A high R^2 in log-space doesn't always translate to small error in original space (due to the non-linear transform). Use error metrics on the original scale (by inverse-transforming predictions).

Log Transforming Features (X)

- Features that are **highly skewed** (especially right-skewed positive variables like income, counts, or sizes) can:
    - Distort distance-based models (KNN, clustering)
    - Break linearity assumptions in linear models
    - Harm performance in gradient-based models (due to unstable gradients)
- Log-transforming can **make relationships more linear** and improve **model fit**.
- Only for **strictly positive, continuous variables** (e.g., income, price, area)

Don't transform features that:

- Contain 0 or negative values (unless using `log1p`)
- Are categorical or ordinal encoded

Check the skewness in the numerical data of the dataframe

```
vars_skewed = df[num_data_features].apply(lambda x:
skew(x)).sort_values(ascending = False)

vars_skewed
```

For columns with `skew > 1`, consider log-transform:

```
# Getting numerical features with skewness higher than 1.
```

```
high_skew = vars_skewed[abs(vars_skewed) > 1]

# Correct the skewness in the numerical features

for feat in high_skew.index:

    df[feat] = np.log1p(df[feat])
```

Scaling / Normalize Feature

Many models (especially those based on distance or gradient descent optimization like linear regression, SVMs, neural nets) benefit from scaling features to a similar range:

Standardization (StandardScaler): Scales features to mean 0 and standard deviation 1 (unit variance). That is, $z = \frac{x - \mu}{\sigma}$. After standardizing, each feature's values are around 0 (most between -3 and 3 if roughly normal). Use when you assume data is normally distributed or for many ML algorithms.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)  # use same scaler fit on
train
```

- StandardScaler is sensitive to outliers (since μ and σ are affected by extreme values)stackoverflow.comgeeksforgeeks.org. If outliers exist, consider robust scaling.

Min-Max Scaling (MinMaxScaler): Rescales features to a given range (default 0 to 1). Formula: $x_{\text{scaled}} = \frac{x - x_{\min}}{x_{\max} - x_{\min}}$. It preserves the shape of the distribution but maps the min of feature to 0 and max to 1stackoverflow.com. Use when you need all features positive and bounded (e.g., some neural network activations) or for algorithms like KNN that might benefit from [0,1] range.

```
from sklearn.preprocessing import MinMaxScaler
mms = MinMaxScaler()
X_train_mm = mms.fit_transform(X_train)
X_test_mm = mms.transform(X_test)
```

- Min-max is **bounded** (no value <0 or >1 after scaling (unless outside train min/max range)) but *not robust to outliers*: an extreme value will compress most data into a tiny rangegeeksforgeeks.org.

RobustScaler: Uses median and IQR for scaling (so outliers have less effect). It centers by median and scales by IQR (75th–25th percentile). Useful if your data has outliers that you don't want to remove but also not let them dominate scaling.

```
from sklearn.preprocessing import RobustScaler
rscaler = RobustScaler()
X_train_rs = rscaler.fit_transform(X_train)
X_test_rs = rscaler.transform(X_test)
```

- **When to scale:** Always scale features **after** train/test split (fit scaler on train only, then transform test) to avoid data leakage. Tree-based models (Decision Trees, Random Forests, XGBoost) **do not require scaling** – they split based on feature thresholds and are invariant to monotonic transformations. But linear models, logistic regression, SVM, KNN, neural nets **do require scaling** for best performance.

- **Don't scale target** (for regression) unless you specifically need to (e.g., if target is extremely skewed, see target transformation below). Also, if a feature is binary 0/1, scaling isn't necessary (though it doesn't hurt either).

Feature Selection

Models Building

Linear Regression

Check Assumptions

Assume we have a feature matrix `X_train, X_test` and target vectors `y_train, y_test` after preprocessing.

Linear Regression (Ordinary Least Squares):

```
from sklearn.linear_model import LinearRegression
lr = LinearRegression()
lr.fit(X_train, y_train)
# Coefficients and intercept:
print("Intercept:", lr.intercept_)
print("Coefficients:", lr.coef_)
```

This fits an OLS linear model: $y = \beta_0 + \beta_1 x_1 + ... + \beta_n x_n$. Coefficients in `lr.coef_` correspond to each feature in `X_train.columns`. Check multicollinearity before interpreting these coefficients too closely. Also, if `X_train` was not scaled and features are on very different scales, the magnitude of coeffs can be misleading. Prediction:

```
y_pred = lr.predict(X_test)
```

- Use `lr.predict` on new data to get predictions.

Ridge Regression: Linear regression with **L2 regularization** (penalty = $\lambda \sum \beta_i^2$). This tends to shrink coefficients towards zero (but not exactly zero) to prevent overfit, especially if features are correlated or $n < p$.

```
from sklearn.linear_model import Ridge
ridge = Ridge(alpha=1.0)  # alpha is the regularization strength λ
ridge.fit(X_train, y_train)
```

- Higher `alpha` means more regularization (coefficients are shrunk more). With `alpha=0`, Ridge becomes OLS. Ridge is good to tackle multicollinearity (it will distribute weights among correlated features) and generally improves prediction when some overfitting is present
  You might tune alpha via cross-validation (use `RidgeCV` or GridSearchCV).

Lasso Regression: Linear regression with **L1 regularization** (penalty = $\lambda \sum |\beta_i|$). Lasso tends to drive some coefficients *exactly* to zero, effectively performing feature selection

```
from sklearn.linear_model import Lasso
lasso = Lasso(alpha=0.1)
lasso.fit(X_train, y_train)
```

- Here too, `alpha` controls regularization strength. Lasso is useful when you suspect many features are irrelevant – it will zero them out if alpha is tuned appropriately. However, Lasso can be unstable if features are highly correlated (it might pick one arbitrarily). Sometimes **Elastic Net** (which combines L1 and L2) is used to balance that.

After fitting all models, compare their performance:

```
# Predict on test data
pred_lr   = lr.predict(X_test)
pred_ridge= ridge.predict(X_test)
pred_lasso= lasso.predict(X_test)
```

We will evaluate these in the next section. But before that, note:

- If features were **scaled**, you can directly compare magnitudes of coefficients. If not, take care in interpreting them.
- For Ridge/Lasso, **feature scaling** is recommended before fitting (so that regularization penalizes all coefficients fairly). If you used StandardScaler earlier, you're fine.
- Regularization introduces bias but can reduce variance. **So on train set, OLS might perform best, but on test, Ridge/Lasso might outperform if overfitting was present.**
- We might want to tune the hyperparameter alpha for Ridge/Lasso. In an interview, you can mention using **cross-validation to find the best alpha** (e.g., using `RidgeCV` or manual GridSearch).

Categorical Model - ANN - XGBoost Classifier - Random Forest

See https://www.kaggle.com/code/faressayah/lending-club-loan-defaulters-prediction#%F0%9F%A4%96-Models-Building

`Roc_auc_score`

Model Evaluation

Regression Evalution

After training models, we need to evaluate how well they perform. For regression tasks, common metrics include **MAE, MSE, RMSE, and R^2**:

- **Mean Absolute Error (MAE):** Average of absolute errors $|y - \hat{y}|$. It's in the same units as the target and is more robust to outliers than MSE (each error contributes linearly). Lower MAE is better.

- **Mean Squared Error (MSE):** Average of squared errors $(y - \hat{y})^2$. Emphasizes larger errors (outliers) due to squaring. Useful mathematically (nice gradients) but less interpretable due to squared units.

- **Root Mean Squared Error (RMSE):** $\sqrt{\text{MSE}}$. This brings error back to original units and is more interpretable (roughly, "on average, our prediction is off by ~RMSE"). It's the square root of average squared errormedium.com. RMSE is often used as it penalizes large errors but still in same unit as y.

- **R^2 (R-squared, Coefficient of Determination):** Proportion of variance in the target explained by the model. $R^2 = 1 - (SS_{res} / SS_{tot})$. R^2 of 1.0 means perfect fit; 0 means model is as good as predicting the mean; negative means model is worse than just predicting mean (this can happen if model is poorly fitted)scikit-learn.org. Higher is better. However, a high R^2 doesn't mean the model is good in all senses – it could be overfitting (check adjusted R^2 or validation performance). For example, **adding more features can only increase R^2, even if they're random noise.**

```
from sklearn.metrics import mean_absolute_error, mean_squared_error,

r2_score

for name, preds in [("Linear Regression", pred_lr),
                    ("Ridge", pred_ridge),
                    ("Lasso", pred_lasso)]:
    print(name)
    print("  MAE:", mean_absolute_error(y_test, preds))
    mse = mean_squared_error(y_test, preds)
    print("  RMSE:", np.sqrt(mse))
    print("  R^2:", r2_score(y_test, preds))
```

Interpretation: If R^2 is 0.88, that means 88% of the variance in y is explained by the modelmedium.com. RMSE of ~4.05 means on average our predictions are about 4 units off the actual. MAE of ~3.5 means median absolute error is 3.5 units.

Cross Validation

Cross-Validation: Instead of relying on a single train-test split (which can be lucky/unlucky), use k-fold cross-validation to more robustly estimate model performance. For example, 5-fold CV will train and test the model 5 times on different splits and give an average score

```
from sklearn.model_selection import cross_val_score
cv_scores = cross_val_score(ridge, X_train, y_train, cv=5,
scoring='neg_root_mean_squared_error')
print("5-fold CV RMSE:", -cv_scores.mean(), "±", cv_scores.std())
```

This will output the mean RMSE across 5 folds. (sklearn returns negative RMSE because it expects higher score = better; we negate to get positive RMSE). You can do this for each model to compare. Usually, one would perform CV on training data for hyperparameter tuning, and final evaluation on test. If test set is small, sometimes cross-validation on all data is used to report an estimate.

Why cross-validate: It gives a better estimate of performance on unseen data by reducing variance due to a particular train-test split. It's especially useful if you have limited data. However, it's computationally heavier (training multiple times). In an interview scenario, you might just show understanding of it rather than actually implement due to time.

Evaluation for classification: (Briefly if needed) – you'd use metrics like accuracy, precision, recall, F1, ROC AUC, etc., and possibly cross_val_score with `scoring='f1'` or so. And confusion matrices.

Appendix

Imbalance (Classification)

Kaggle:
https://www.kaggle.com/code/janiobachmann/credit-fraud-dealing-with-imbalanced-datasets

If the dataset is for **classification** and classes are imbalanced (e.g., 90% of data is class A, 10% class B), special techniques are needed so that the model doesn't just predict the majority class always.

- **Check imbalance:** `df['target'].value_counts()` to see distribution. If one class is overwhelmingly frequent, standard training may yield a model with high accuracy that ignores the minority class. Use metrics like precision, recall, F1 instead of accuracy in such cases.

- **Resampling Techniques:**

SMOTE (Synthetic Minority Over-sampling Technique): A popular method to generate synthetic samples for the minority class by interpolating between existing minority instancesgeeksforgeeks.orggeeksforgeeks.org. This adds new "similar" points rather than just duplicating originals.

```
from imblearn.over_sampling import SMOTE
X_res, y_res = SMOTE().fit_resample(X_train, y_train)
```

    - Now `X_res, y_res` have an increased number of minority class samples (balanced). Train the model on this resampled data. *Note:* Only resample the training set (never the test).

    - **Random Oversampling:** Simply duplicate some minority class examples until balance is achieved. (SMOTE is usually preferable to avoid exact duplicates and overfitting.)

    - **Random Undersampling:** Remove some majority class samples to balance. Effective if you have plenty of data, but you lose information from the majority class. Can be okay if majority class is extremely large.

    - **Combination (SMOTE + Tomek links/ENN):** Oversample and then clean noisy overlaps (beyond scope of an interview unless specifically asked).

- **Algorithmic approaches:**

    - **Class weights:** Many classifiers (e.g., `sklearn.linear_model.LogisticRegression` or `sklearn.ensemble.RandomForestClassifier`) have a `class_weight` parameter. `class_weight='balanced'` will automatically weight classes inversely proportional to their frequency, so the model "pays more attention" to minority class. This is often a quick remedy.

    - **Threshold moving:** If using a probability model, you can adjust the decision threshold to favor the minority class (lower the threshold for positive prediction to catch more of minority class).

    - **Ensemble methods:** Some approaches like EasyEnsemble train multiple models on different balanced subsets.

- **SMOTE example:** Suppose we have a 1:9 minority:majority ratio. SMOTE will synthetically generate minority samples until the ratio is 1:1 (or whatever you specify). This **addresses class imbalance by adding new minority examples**geeksforgeeks.org. Always check that the synthetic data makes sense and doesn't introduce noise – typically SMOTE is reliable for numeric feature spaces.

- **Evaluation:** Use metrics like **ROC AUC**, **F1-score**, **precision/recall** to evaluate performance on imbalanced data. Accuracy is misleading (e.g., 90% accuracy by always predicting the majority class in a 90/10 split, but that model is useless for minority). A confusion matrix is helpful to see if minority class is being predicted at all.

Numerical to Categorical (Binning)

Sometimes a numeric variable should be treated as categorical, or you want to reduce its complexity by binning. Examples: age into age groups, income brackets, etc., or a year variable that actually denotes a category (e.g., Year of make of a car might be better as categorical if you don't want to assume linear progression).

pd.cut: Use for defined bins. You specify the bin edges (or number of bins) and it buckets the values.

```
# Example: bin ages into groups
bins = [0, 12, 19, 35, 60, np.inf]
labels = ['Child', 'Teen', 'Young_Adult', 'Adult', 'Senior']
df['AgeGroup'] = pd.cut(df['Age'], bins=bins, labels=labels)
```

- This will create a new categorical column *AgeGroup*. Values <=12 become "Child", 13-19 "Teen", etc., and above 60 "Senior". `np.inf` indicates an open-ended last bin.

- **pd.qcut:** Use for quantile-based bins. It will try to assign values so that each bin has (roughly) equal number of observations. For example, `pd.qcut(df['Income'], q=4, labels=['Q1','Q2','Q3','Q4'])` splits income into quartiles by value.

- **When to bin:** Binning can make sense if the numeric relationship is non-linear or if you want to reduce noise by grouping. But it also throws away some information (the exact value within a bin). It's useful in models that can't handle non-linear relations well or for readability of results. In linear regression, for instance, a non-linear effect might be better captured by binning a feature.

- **Convert numeric IDs to categorical:** If you have identifiers like CustomerID or ProductID that are numeric but just identifiers, treat them as categoricals (or more often, drop them if they are just IDs with no predictive value). You can convert by `df['ID'] = df['ID'].astype('category')` in pandas, or simply one-hot encode if needed.

Linear Regression Assumption

Linear regression (and to an extent any OLS-based model) relies on several key assumptions. It's good practice to check these assumptions on the **training data's residuals**:

1. **Linearity:** The relationship between each independent variable and the dependent variable is linear (additive). If the true relationship is non-linear, the linear model will struggle.
   How to check: Plot **residuals vs. fitted values**. If the model is appropriate, residuals should scatter randomly around 0 with no obvious patternbookdown.org. If you see a curve or U-shape in residuals, it indicates non-linearity (e.g., you might need a quadratic term or another model)bookdown.org.
   Also, scatterplots of `y` vs each `x` before modeling can suggest if a transform is needed (like log x or x^2) to achieve linearity.

2. **Independence of errors:** Residuals should be independent of each other. This is especially an issue with time-series data or any data with inherent ordering (e.g., consecutive samples from same source). Non-independence (autocorrelation) means your confidence intervals and tests are off.
   How to check: If data is time-ordered, plot residuals over time (or index). They should not show patterns (like cycles or trends). A **Durbin-Watson test** can statistically check autocorrelation of residuals (DW ≈ 2 means no autocorrelation, <<2 or >>2 indicates positive or negative autocorrelation)medium.com. In Python: `import statsmodels.api as sm; sm.stats.durbin_watson(residuals)`.
   For i.i.d. data (no time component), independence is usually assumed from the data collection design. If you have groups (e.g., multiple rows per user), that violates independence unless you use a grouped model (mixed effects).

3. **Homoscedasticity (Constant Variance of errors):** The residuals have constant variance at every level of the independent variables. In other words, the scatter of residuals should be roughly the same across all fitted values.
   How to check: Again, **residuals vs fitted plot** is key. If you see the residuals' spread increasing or decreasing with fitted values (a "fanning" shape), that's heteroscedasticitybookdown.org. For example, residuals get larger for larger predictions – common if y increases in variance with its mean (e.g., heteroscedastic nature).
   You can also plot residuals vs each predictor to see if variance changes with X.
   Fixes: A log or Box-Cox transform of y can often stabilize variance (turn heteroscedasticity into homoscedastic). There are also techniques like weighted least squares if variance of residuals can be predicted by some factor.

Normality of errors: Residuals should be (approximately) normally distributed (especially for inference – t-tests, CIs on coefficients – to be valid). This matters less for prediction accuracy (Central Limit Theorem often ensures approximate normality if sample size large), but it's an assumption for the classical OLS inference.
How to check: Plot a **QQ-plot** (quantile-quantile plot) of residuals vs theoretical normal distributionbookdown.org. Ideally, points lie on the 45° line. Systematic deviations (like an S-shaped curve) mean residuals are skewed or have heavy tailslibrary.virginia.edu. Also look at a histogram of residuals to see if roughly bell-shaped.
Example using SciPy:

```
import scipy.stats as stats
stats.probplot(residuals, dist="norm", plot=plt)
plt.title("Normal QQ-plot of residuals")
plt.show()
```

4. If residuals are not normal: If skewed, a target transform (log) might help as mentioned. If there are outliers causing non-normality, address those. Minor deviations are usually fine if your goal is prediction.

- **Mean of residuals = 0:** In a regression with intercept, this is usually inherently satisfied (the model ensures residuals sum to zero). Just ensure you included an intercept term in your model (sklearn LinearRegression includes it by default). If not, residuals may have non-zero mean, indicating bias.

When these assumptions hold, your linear regression is likely appropriate. If violated:

- Non-linearity suggests adding polynomial terms or using a non-linear model.

- Heteroscedasticity suggests transforming y or using robust standard errors.

- Autocorrelation suggests using time-series models or adding lags.
- Multicollinearity (not an assumption of linear regression per se, but an issue) we addressed earlier.

Assumption Check Example:
After fitting a linear model, say `preds = lr.predict(X_train)` and `res = y_train - preds`. You'd do:

```
plt.scatter(preds, res)
plt.axhline(0, color='red', ls='--')
plt.xlabel("Fitted values"); plt.ylabel("Residuals")
plt.title("Residuals vs Fitted")
plt.show()
```

You want to see a horizontal cloud of points. No patterns (like U-shape or expanding funnel)
Next:

```
import scipy.stats as stats
stats.probplot(res, plot=plt)
plt.title("QQ-plot of residuals")
plt.show()
```

You want an approximate straight line. Minor deviations at ends are okay; major curvature is a concern.

High-level data processing procedures

For data (not time series)

1. have a clear problem defined. Such as what are the variables, what is the topic we want to investigate
2. remove unuseful variables
3. check variable types

if numeric:

- check variable distribution
- if it is skew

- check extreme values/outliers

if categorical

- check distribution of the categories, if it is balanced
- check if the categorical variables are ordinal, or just categoriories without relationship
- may convert to dummy variable or numeric ordinal variable

if datetime

- convert to number of days

4. check null values of each variable

- If too many NAs, such as > 90%, we can directly drop it
- But for half of missing values, we should not directly drop it. If the variable is important, such value is missing also implied some information. We can create a new column of indicators to show whether or not it is missing. We need to investigate more about specify variable
- for less NAs, either drop rows or fillna with mean/median, depending on meaning of variable

5. Others …. standardize etc
6. If we want to run mode, check assumptions

Time series

1. check stationary
2. check autocorrelation
3. check outliers