Reading the Dataset

```
df = pd.read_csv("data.csv")
df.shape      # (rows, columns)
df.info()   # concise summary (types and non-null counts)

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

（如果没有header：df = pd.read_csv('data.csv', header=None)  # 告诉 pandas 第一行不是列名
df.columns = ['col1', 'col2', 'col3', 'col4', 'y']  # 手动加 header）
```

Data Exploration

Missing Values

Looks like there's some missing values. I think we can then handle the missing values.

- **Check Missing values:** `print(df.isnull().sum())`
- **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.
- missing value也可能是有价值的

Mean/Median Imputation: Replace missing numeric values with the mean or median of that column.

```
df['Col'] = df['Col'].fillna(df['Col'].median())
```

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')
```

Plot

Time-series

Visualize the overall trend and detect seasonality or anomalies. Start with a simple line plot

```
import matplotlib.pyplot as plt
```

```
plt.figure(figsize=(12, 5))
plt.plot(df['date'], df['value'])
plt.title("Raw Time Series")
plt.xlabel("Date")
plt.ylabel("Value")
plt.show()
```

“Then I visualize each feature’s distribution to see their distributions. I use histograms first, and then for their relationships with Y I also use scatter plots to see”

Divide into numerical / categorical

```
X = df.drop(columns=['y'])
num_cols = [“X1”, “X2”]

cat_cols = ["credit.policy", "purpose", "inq.last.6mths", "delinq.2yrs",
"not.fully.paid"]
cat_data = X[cat_cols]

import matplotlib.pyplot as plt

import seaborn as sns

# 1️⃣ numerical

df.hist()

plt.show()

for i in num_cols:
 sns.histplot(df[i], bins=20, kde=True)
 plt.show()

for c in num_cols:
    sns.scatterplot(x=df[c], y=df["y"], s=20, alpha=0.6)
    sns.regplot(x=df[c], y=df["y"], scatter=False, ci=None, color="black")
        plt.title(f"{c} vs y");
    plt.show()


# 2️⃣ categorical
```

- df.nunique() or value counts with df['col'].value_counts()
    - check Imbalance

```
for col in cat_cols:
    print(df[col].value_counts(normalize=True))
    sns.countplot(x=col, data=df)
```

```
    plt.show()
```

Outliers

“Then I’ll check for outliers. I think we can use a boxplots to flag and understand the outliers. “

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

```
num_df.boxplot()
```

```
for col in num_cols:
    sns.boxplot(y = df[col])
    plt.show()
```

(
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

```
def iqr_bounds(df, x_col):
      Q1 = df[x_col].quantile(0.25)
      Q3 = df[x_col].quantile(0.75)
      IQR = Q3 - Q1
      lower_bound = Q1 - 1.5 * IQR
      upper_bound = Q3 + 1.5 * IQR
      outliers = df[(df[x_col] < lower_bound) | (df[x_col] > upper_bound)]

for c in num_cols:
    print(iqr_bounds(df, c)
```

)

How to handle 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.

- 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(logwer = df[var].quantile(.05), upper =

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

[TODO] Time-Series Outliers

[TODO] Other Basic Statistics

df.describe()

time-series

Understand overall distribution, volatility, and stability.

```
# Rolling statistics
df['rolling_mean'] = df['value'].rolling(window=7).mean()
df['rolling_std'] = df['value'].rolling(window=7).std()

plt.figure(figsize=(12,5))
plt.plot(df['date'], df['value'], label='Original')
plt.plot(df['date'], df['rolling_mean'], label='7-day Mean')
plt.plot(df['date'], df['rolling_std'], label='7-day Std')
plt.legend()
plt.show()
```

Feature Engineering

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

- **Extract components**: day of week, hour, month, etc.
- **Convert to time since event**: e.g., “days since last login”

For Time Series:

- **Lag Features**: previous values (`sales_t-1`)
- **Rolling Mean/Std**: smoothing over past few time steps
- **Difference**: to remove trends

Skew / Log

Plot check target variable:

```
# Plot histogram of House selling prices

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

plt.title('House Prices Histogram')

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).

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).

- **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.

Log Transforming Features (X)

- Log-transforming can **make relationships more linear** and improve **model fit**.
- Only for **strictly positive, continuous variables** (e.g., income, price, area)

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:

```
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). 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 1. 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 range

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).

Time-Series

Stationary

“To test stationarity, I first plot the time series and check the rolling mean and variance. Then I run the ADF and KPSS tests — ADF tests for a unit root and KPSS tests for level stationarity. If the series is non-stationary, I apply transformations like differencing for trend, seasonal differencing for seasonality, and log or Box–Cox for variance stabilization.
I repeat until both tests confirm stationarity, which is critical before fitting ARIMA models.”

A stationary time series has constant mean, variance, and autocovariance over time.
In other words, its statistical properties don’t depend on time.

Detect:

1. Plot the series → look for trend or seasonality.

Plot rolling mean and rolling std:

```
rolmean = ts.rolling(12).mean()
rolstd = ts.rolling(12).std()
plt.plot(ts, label='Original')
plt.plot(rolmean, label='Rolling Mean')
plt.plot(rolstd, label='Rolling Std')
➜ If they change over time → non-stationary.
```

2️⃣ Statistical Tests

Test	Null Hypothesis	Reject H_0 →	Python Function
ADF (Augmented Dickey–Fuller)	Series has a unit root (non-stationary)	Stationary	`adfuller()`
KPSS (Kwiatkowski–Phillips–Schmidt–Shin)	Series is stationary	Non-stationary	`kpss()`

🧩 **Interpret together:**

- ADF rejects H_0, KPSS doesn't → **stationary**
- Both fail → **non-stationary**
- Both reject → **trend-stationary** (needs detrending)

3️⃣ Autocorrelation Plots

- Use **ACF/PACF**: slowly decaying ACF → non-stationary.

🧰 Making Data Stationary

Problem	Fix	Python
Trend (mean changes)	Differencing: `ts.diff()`	`python ts_diff = ts.diff().dropna()`
Seasonality	Seasonal differencing: `ts.diff(period)`	`ts.diff(12)`
Changing variance	Log / Box-Cox transform	`np.log(ts)` or `boxcox(ts)`
Deterministic trend	Detrending (subtract fitted line)	`ts - sm.OLS(ts, t).fit().fittedvalues`
Seasonal + Trend	Seasonal decomposition	`seasonal_decompose(ts)`

[TODO] autocorrelation

Identify lag relationships and potential seasonal effects.

Create features

Generating calendar-based features, lags, and rolling stats as useful predictors for forecasting models.

```
df['day_of_week'] = df['date'].dt.dayofweek
df['month'] = df['date'].dt.month
df['lag_1'] = df['value'].shift(1)
df['rolling_7'] = df['value'].rolling(window=7).mean()
df['expanding_mean'] = df['value'].expanding().mean()
```

Correlation Test

```
import seaborn as sns
# Correlation between lag features
sns.heatmap(df[['value', 'lag_1', 'rolling_7']].corr(), annot=True)
plt.show()
```

[如果有] Encoding Categorical Variables

One-Hot Encoding:

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

- **Note:** One-hot encoding increases dimensionality

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))
```

- **Be cautious:** Label encoding implies an ordinal relationship (2 > 1 > 0). Only use if the algorithm is tree-based or if the categories are ordinal (e.g., "Low", "Medium", "High")

Correlation/Multicollinearity

Multicollinearity means two or more features are highly correlated, which can cause unstable result on linear model coefficients (inflating standard errors, making it hard to interpret coefficients)

```
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)

Variance Inflation Factor (VIF): 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)

```
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
```

Handling multicollinearity:

- If two features are very correlated (say r > 0.9), you can drop one without losing much information
- For modeling, using **Ridge regression** (L2 regularization) can mitigate multicollinearity by shrinking coefficients, and **Lasso** (L1) can even drop one of the correlated features.
- PCA

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):

```
import numpy as np
import statsmodels.api as sm

from sklearn.metrics import mean_squared_error, r2_score

X_train_const = sm.add_constant(X_train)
# X_test_const = sm.add_constant(X_test)

model = sm.OLS(y_train, X_train_const)
results = model.fit()
```

This fits an OLS linear model: $y = \beta_0 + \beta_1 x_1 + ... + \beta_n x_n$.

```
print(results.summary()) #包含pvalues
```

```
y_pred = results.predict(X_test_const)
```

analysis

```
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
r2 = r2_score(y_test, y_pred)
print(f"Test RMSE: {rmse:.3f}")
print(f"Test R²: {r2:.3f}")
```

Or Sklearn:

```
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_)
```
Check multicollinearity before interpreting these coefficients.

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

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.

Ridge/Lasso

Reduce model complexity and prevent over-fitting.
Ridge vs. Lasso: The key difference between these two is the penalty term.
Lasso - Not only helps in reducing over-fitting but also it can help us in feature selection

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. **cross-validation to find the best alpha** (e.g., using `RidgeCV` or manual GridSearch).

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)
```

- `alpha` controls regularization strength. **cross-validation to find the best alpha** (e.g., using `RidgeCV` or manual GridSearch).

- 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.**

Regression Evaluation

“We can evaluate the model using R², RMSE, and residual plots. “

For regression tasks, common metrics include **MAE, MSE, RMSE, and R²**:

- **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.

- **R² (R-squared, Coefficient of Determination):** Proportion of variance in the target explained by the model. R² = 1 - (SS_res / SS_tot). R² 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 R² doesn’t mean the model is good in all senses – it could be overfitting (check adjusted R² or validation performance). For example, **adding more features can only increase R², 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: “the R^2 is 0.88, that means 88% of the variance in y is explained by the model. 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.”

Residual Analysis

Plot residuals vs. predicted values

- Should look like random noise → indicates good fit
- If you see patterns (e.g., curves or heteroscedasticity), model may be mis-specified

Look at **QQ plot** of residuals to check normality (assumption of linear regression)

Example: “the residuals showed slight heteroscedasticity — suggesting we might improve with feature transformation or WLS”

Interpret Coefficient - Check Feature Coefficients

Look for unexpected signs or very small coefficients → could indicate multicollinearity or weak predictors

Where to Improve

“Next steps could include Lasso regression to reduce overfitting and improve interpretability.”

a. Model Fit

- Consider regularization (Ridge/Lasso) if overfitting or multicollinearity
- Try adding interaction terms or polynomial features if linearity doesn’t hold

b. Features

- Check correlation heatmap and VIF for multicollinearity
- Feature engineering: Are there important variables missing? Can you transform existing ones (e.g., log)?

Feature Selection

Use statistical techniques to evaluate the relationship between each feature and the target variable.

- **Univariate Statistics**
  - For classification: `chi-squared`, `ANOVA F-test`, `mutual information`
  - For regression: `Pearson correlation`, `mutual information`
- **Variance Thresholding**: Remove features with low variance.
- **Correlation Filtering**: Remove one of each pair of highly correlated features (e.g., `|corr| > 0.9`).

1. Embedded method:

"These do feature selection as part of training the model. For example, Lasso regression can shrink some feature coefficients to zero, and tree-based models like Random Forest naturally give you feature importances."

Pros: Efficient, captures interactions
Cons: Depends on model choice

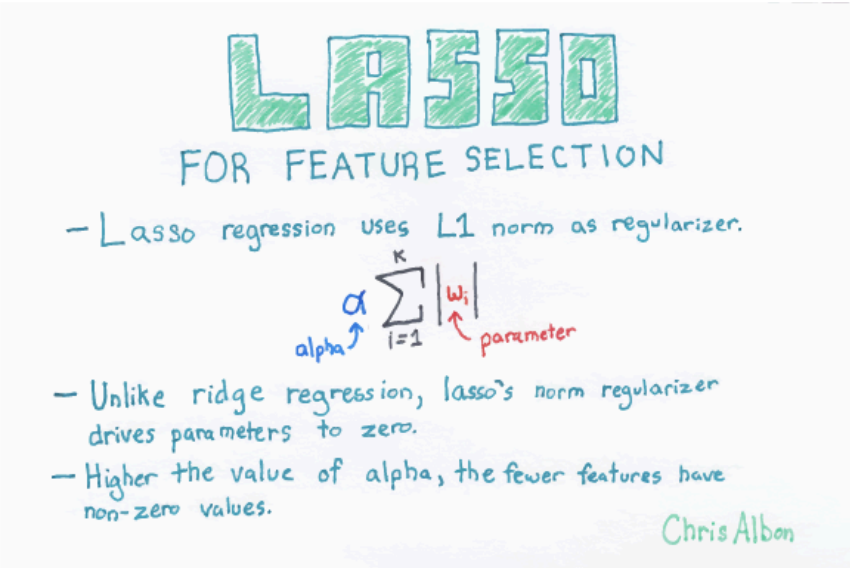
```
from sklearn.linear_model import LassoCV
# Example: Embedded
model = LassoCV().fit(X, y)
selected_features = X.columns[model.coef_ != 0]

from sklearn.ensemble import RandomForestClassifier
# Example: Tree-based Embedded
model = RandomForestClassifier().fit(X, y)
importances = model.feature_importances_
```

`the model decided that` **`no predictor contributes enough signal relative to the penalty (alpha).`**

Feature selection is built into the model training process.

- **L1 Regularization (Lasso Regression)**: Shrinks some coefficients to zero.
- **Since proTree-based Feature Importance**: Use `Random Forest`, `XGBoost`, or `LightGBM` feature importances.
  - "Tree-based models like Random Forest or XGBoost are great for embedded feature selection because they naturally rank feature importance based on how useful each feature is for splitting the data during training."
- **Elastic Net**: Combines L1 and L2 regularization.



Tree-based model like RF. We use RandomForest to select features based on feature importance. We calculate feature importance using node impurities in each decision tree. In Random forest, the final feature importance is the average of all decision tree feature importance.used a LightGBM. Or an XGBoost object as long it has a feature_importances_ attribute

“In practice, I usually start with a filter or embedded method for a quick pass, and if needed, try a wrapper method for fine-tuning.”

Other Models

Or do you want me to try run again with Lasso a alpha?
Ridge

LightGbm Model

`LightGBM` is fast and great for structured/tabular data.

```
import lightgbm as lgb

# Create LightGBM dataset
```

```
train_data = lgb.Dataset(X_train, label=y_train)
test_data = lgb.Dataset(X_test, label=y_test, reference=train_data)

# Define parameters
params = {
    'objective': 'binary',
    'metric': 'binary_logloss',
    'verbosity': -1,
}

# Train model
gbm = lgb.train(params, train_data, valid_sets=[test_data],
num_boost_round=100, early_stopping_rounds=10, verbose_eval=False)

# Predict
y_pred_prob = gbm.predict(X_test)
y_pred = (y_pred_prob > 0.5).astype(int)
print("Accuracy:", accuracy_score(y_test, y_pred))
```

Random Forest

```
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
import matplotlib.pyplot as plt

# Generate synthetic dataset
np.random.seed(42)
X = np.random.rand(100, 3)  # 3 features
true_coeffs = np.array([5, -3, 2])
y = X @ true_coeffs + np.random.normal(0, 0.5, 100)  # y = 5x1 - 3x2 + 2x3 +
noise
# Train/test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Train Random Forest
model = RandomForestRegressor(n_estimators=100, random_state=42)
```

```
model.fit(X_train, y_train)

# Predict and evaluate
y_pred = model.predict(X_test)
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
r2 = r2_score(y_test, y_pred)

print(f"RMSE: {rmse:.3f}")
print(f"R²: {r2:.3f}")
```

Evaluation Random Forest

Random Forest is an ensemble of decision trees, so it's nonlinear and non-parametric, which means:

- You can't interpret coefficients like in linear models
- But you can interpret:

Feature Importance

- Most accessible interpretation method
- Measures how much each feature **reduces impurity (Gini or MSE)** across all trees

```
Model.feature_importances_
```

"Feature importance showed that variables like `xxx` and `xxx` were key predictors"

For Regression:

- **RMSE**, **MAE**: average prediction error
- **R²**: percentage of variance explained

For Classification:

- **Accuracy**, **Precision/Recall**, **F1-score**, **ROC AUC**
- **Confusion Matrix**: understand false positives/negatives

Improve a Random Forest

1. Hyperparameter Tuning

Try using `GridSearchCV` or `RandomizedSearchCV` to tune:

- `n_estimators` (number of trees)
- `max_depth` (tree size)
- `max_features` (features considered per split)

- `min_samples_leaf` (regularization)

  "Our initial model used default parameters. We can likely improve performance with a deeper grid search on max_depth and number of estimators."

2. Overfitting

- Check **train vs. test error**
- Try reducing `max_depth`, increasing `min_samples_leaf`, or using fewer trees

3. Feature Engineering

Even tree-based models benefit from:

- Creating interaction features
- Binning or categorizing important numeric values
- Encoding rare categories properly

LightGBM

Feature importance

```
import lightgbm as lgb
lgb.plot_importance(model, max_num_features=20)
```

—————————————

Evaluate Using Common Metrics

```
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
import numpy as np

mae = mean_absolute_error(y_test, y_pred)
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
r2 = r2_score(y_test, y_pred)

print(f"MAE: {mae:.4f}")
print(f"RMSE: {rmse:.4f}")
print(f"R²: {r2:.4f}")
```

———-

Plot Predicted vs Actual Values

```
import matplotlib.pyplot as plt
```

```
plt.scatter(y_test, y_pred, alpha=0.5)
plt.xlabel("Actual")
plt.ylabel("Predicted")
plt.title("Actual vs. Predicted")
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()],
color="red", lw=2)
plt.show()
```

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.

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)
OR: train, test = train_test_split(data, test_size=0.33, random_state=42)
```

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 pattern. 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).
   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). 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 heteroscedasticity. 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

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.