Laptop Price Prediction - Full Repro Pipeline

This notebook contains a reproducible end-to-end pipeline for the **Laptop Price Prediction** project: data loading, cleaning, feature engineering, EDA, modeling (Linear / RandomForest / GradientBoosting), light hyperparameter tuning, evaluation, feature importance, saving the model, and a predict function.

Files produced by running this notebook:

- `best_laptop_price_model.pkl` (saved model)

  Note: Update the `DATA_PATH` variable below to point to your `laptop.csv` if different.

```
# Setup - imports and data path
DATA_PATH = "laptop.csv"  # change if needed
MODEL_OUT = "best_laptop_price_model.pkl"

import pandas as pd, numpy as np, re, joblib
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor,
GradientBoostingRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error,
r2_score
import matplotlib.pyplot as plt
%matplotlib inline


# Load data
df = pd.read_csv(DATA_PATH)
print("Shape:", df.shape)
df.head()
```

Cleaning & Feature Engineering

- Convert `Ram`, `Inches`, `Weight` to numeric
- Parse `ScreenResolution` to extract `X_res`, `Y_res`, `Touchscreen`, `IPS` and compute `PPI`
- Parse `Memory` into `Storage_GB`, `SSD_GB`, `HDD_GB`, `Flash_GB`, `eMMC_GB`
- Extract `Cpu_brand`, `Gpu_brand`
- Group `OpSys` into `OpSys_group`

```
def extract_numeric(val):
    try:
        s = str(val)
        m = re.search(r"[\d.]+", s)
        return float(m.group(0)) if m else np.nan
    except:
        return np.nan

def parse_resolution(res):
    s = str(res)
    touchscreen = 1 if "Touchscreen" in s or "touchscreen" in s else 0
    ips = 1 if "IPS" in s or "ips" in s else 0
    m = re.search(r"(\d{3,4})\s*x\s*(\d{3,4})", s)
    if not m: m = re.search(r"(\d{3,4})x(\d{3,4})", s)
    if m: x = int(m.group(1)); y = int(m.group(2))
    else: x = np.nan; y = np.nan
    return pd.Series([touchscreen, ips, x, y])

def cpu_brand(cpu):
    s = str(cpu); tokens = s.split(); return tokens[0] if
len(tokens)>0 else "Unknown"

def parse_memory(mem):
    s = str(mem)
    parts = re.split(r'\s*\+\s*', s)
    total = 0.0; ssd=0; hdd=0; flash=0; eMMC=0
    for p in parts:
        m_tb = re.search(r'(\d+\.?\d*)\s*TB', p, re.IGNORECASE)
        m_gb = re.search(r'(\d+\.?\d*)\s*GB', p, re.IGNORECASE)
        if m_tb: gb = float(m_tb.group(1)) * 1024
        elif m_gb: gb = float(m_gb.group(1))
        else: gb = 0.0
        total += gb
        if re.search(r'ssd', p, re.IGNORECASE): ssd += gb
        if re.search(r'hdd', p, re.IGNORECASE): hdd += gb
        if re.search(r'flash', p, re.IGNORECASE): flash += gb
        if re.search(r'eMMC', p, re.IGNORECASE): eMMC += gb
    return pd.Series([total, ssd, hdd, flash, eMMC])

def opsys_group(x):
    s = str(x)
    if "Windows" in s or "windows" in s: return "Windows"
    if "macOS" in s or "Mac OS" in s or "mac" in s: return "macOS"
    if "Linux" in s or "linux" in s: return "Linux"
    if "No OS" in s or "NoOS" in s or "no os" in s: return "No OS"
    return "Other"

# Apply transformations
df = df.copy()
```

```
# find price column and rename to Price
price_col = [c for c in df.columns if "price" in c.lower()]
if len(price_col)==0:
    raise ValueError("No price column found.")
df = df.rename(columns={price_col[0]: "Price"})

df['Inches'] = df['Inches'].apply(extract_numeric)
df[['Touchscreen','IPS','X_res','Y_res']] =
df['ScreenResolution'].apply(parse_resolution)
df['PPI'] = ((df['X_res']**2 + df['Y_res']**2)**0.5) / df['Inches']
df['Ram'] = df['Ram'].astype(str).str.replace('GB','',regex=False)
df['Ram'] = pd.to_numeric(df['Ram'], errors='coerce')
df['Weight'] =
df['Weight'].astype(str).str.replace('kg','',regex=False).str.replace(
'kgs','',regex=False)
df['Weight'] = df['Weight'].replace('?', np.nan)
df['Weight'] = pd.to_numeric(df['Weight'], errors='coerce')
df['Cpu_brand'] = df['Cpu'].apply(cpu_brand)
df['Gpu_brand'] = df['Gpu'].apply(lambda x: str(x).split()[0])
df[['Storage_GB','SSD_GB','HDD_GB','Flash_GB','eMMC_GB']] =
df['Memory'].apply(parse_memory)
df['OpSys_group'] = df['OpSys'].apply(opsys_group)

# Drop columns not used directly
df_model =
df.drop(columns=['ScreenResolution','Cpu','Memory','Gpu','OpSys'])
# Drop rows missing critical features
df_model =
df_model.dropna(subset=['Price','Inches','X_res','Y_res','Ram','Storag
e_GB'])

print("After cleaning shape:", df_model.shape)
df_model.head()
```

Quick EDA: price distribution, top companies, price vs RAM and PPI

```

# Price distribution
price_col = 'Price'
plt.figure(figsize=(8,4))
plt.hist(df_model[price_col], bins=50)
plt.title("Price distribution")
plt.xlabel("Price")
plt.ylabel("Count")
plt.show()

# Average price by company (top 15)
avg_price_company = df_model.groupby('Company')
```

```
['Price'].mean().sort_values(ascending=False).head(15)
plt.figure(figsize=(10,4)); avg_price_company.plot(kind='bar');
plt.title("Avg Price by Company (top 15)"); plt.ylabel("Avg Price");
plt.show()

# Median price by Ram
median_price_ram = df_model.groupby('Ram')
['Price'].median().sort_index()
plt.figure(figsize=(8,4)); plt.plot(median_price_ram.index,
median_price_ram.values, marker='o'); plt.title("Median Price by
Ram"); plt.xlabel("RAM (GB)"); plt.ylabel("Median Price"); plt.show()

# Price vs PPI scatter
plt.figure(figsize=(8,4)); plt.scatter(df_model['PPI'],
df_model['Price'], alpha=0.6); plt.title("Price vs PPI");
plt.xlabel("PPI"); plt.ylabel("Price"); plt.show()
```

Modeling: preprocessing pipeline + baseline models (Linear, RF, GB) + light tuning

```
# Features and target
X = df_model.drop(columns=['Price'])
y = df_model['Price'].astype(float)

numeric_features =
['Inches','Ram','Weight','PPI','Storage_GB','SSD_GB','HDD_GB','Flash_G
B','eMMC_GB','X_res','Y_res']
numeric_transformer = Pipeline(steps=[('imputer',
SimpleImputer(strategy='median')), ('scaler', StandardScaler())])
categorical_features =
['Company','TypeName','Cpu_brand','Gpu_brand','OpSys_group']
categorical_transformer = Pipeline(steps=[('imputer',
SimpleImputer(strategy='constant', fill_value='missing')), ('onehot',
OneHotEncoder(handle_unknown='ignore'))])
preprocessor = ColumnTransformer(transformers=[('num',
numeric_transformer, numeric_features), ('cat',
categorical_transformer, categorical_features)])

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

models = {
    "LinearRegression": Pipeline(steps=[('pre', preprocessor), ('reg',
LinearRegression())]),
    "RandomForest": Pipeline(steps=[('pre', preprocessor), ('reg',
RandomForestRegressor(random_state=42, n_jobs=1))]),
    "GradientBoosting": Pipeline(steps=[('pre', preprocessor), ('reg',
GradientBoostingRegressor(random_state=42))])
```

```
}

results = []
for name, model in models.items():
    model.fit(X_train, y_train)
    preds = model.predict(X_test)
    results.append({"model": name, "rmse": mean_squared_error(y_test,
preds, squared=False), "mae": mean_absolute_error(y_test, preds),
"r2": r2_score(y_test, preds)})

import pandas as pd
pd.DataFrame(results).sort_values('rmse')


# Light Grid Search for RandomForest (small grid - quick)
rf_pipeline = models['RandomForest']
rf_param_grid = {'reg__n_estimators': [100], 'reg__max_depth': [10,
None]}
rf_gs = GridSearchCV(rf_pipeline, rf_param_grid, cv=3,
scoring='neg_root_mean_squared_error', n_jobs=1)
rf_gs.fit(X_train, y_train)
best_rf = rf_gs.best_estimator_

# Evaluate tuned RF
preds = best_rf.predict(X_test)
print("RF Tuned RMSE:", mean_squared_error(y_test, preds,
squared=False))
print("RF Tuned MAE:", mean_absolute_error(y_test, preds))
print("RF Tuned R2:", r2_score(y_test, preds))


# Feature importance (for tree-based model)
preprocessor.fit(X_train)
num_names = numeric_features
cat_cols =
preprocessor.named_transformers_['cat'].named_steps['onehot'].get_feat
ure_names_out(categorical_features)
feature_names = list(num_names) + list(cat_cols)

try:
    importances = best_rf.named_steps['reg'].feature_importances_
    feat_imp = pd.DataFrame({"feature": feature_names, "importance":
importances}).sort_values('importance', ascending=False).head(20)
    display(feat_imp)
    plt.figure(figsize=(8,4))
    plt.barh(feat_imp['feature'].head(10)[::-1],
feat_imp['importance'].head(10)[::-1])
    plt.title("Top 10 Feature Importances")
    plt.xlabel("Importance")
    plt.show()
```

```
except Exception as e:
    print("Feature importances not available:", e)


# Save best model
joblib.dump(best_rf, MODEL_OUT)
print("Saved model to", MODEL_OUT)


# Predict function example
def predict_price(sample_dict):
    sample_df = pd.DataFrame([sample_dict])
    missing = set(X.columns) - set(sample_df.columns)
    for m in missing: sample_df[m] = np.nan
    pred = best_rf.predict(sample_df)[0]
    return float(pred)

# Example: use a row from test set
example = X_test.iloc[0].to_dict()
print("True price:", float(y_test.iloc[0]))
print("Predicted price:", predict_price(example))
```