ONE DAY AI AND DATA SCIENCE WORKSHOP (GOOGLE COLAB)

Customer shopping behaviour analysis, end to end

Who this is for

Business leaders, developers, and consultants who want a reusable, step by step workflow to apply AI/ML/DL to a tabular dataset and translate results into decisions.

What you will leave with

- A repeatable Colab notebook workflow you can reuse on a new dataset tomorrow

- A cleaned CSV, an ML-ready encoded CSV, a clustered CSV, and optional saved models

- A set of EDA outputs you can copy into a short stakeholder report

Important note about this dataset

This file has no date column and almost one row per customer. That means:

- Forecasting next month orders is not possible from this file alone

- Association rules will not produce meaningful cross sell rules unless you have baskets or repeat purchases

This lab includes checks and shows what extra data you need.

Dataset

shopping_behavior_updated (1).csv

Session structure (suggested)

1) Setup and load data

2) EDA (quality, distributions, segment comparisons, pivots, correlations, outliers)

3) Cleaning and feature engineering

4) Regression (predict spend) including an MLP baseline

5) Classification (predict category), confusion matrix, tuning, MLP baseline

6) Customer segmentation (clustering) and actions

7) Association rules concept and suitability check

8) Forecasting concept and LSTM demo (with a clear note about required data)

9) Export and sharing

==========

SECTION 1: COLAB SETUP AND LOADING

==========

CELL 1: Environment check

Why: confirms your runtime and working folder.

```
import os, sys
print("Python version:", sys.version.split()[0])
print("Current folder:", os.getcwd())
print("Files here:", os.listdir(".")[:20])
```

CELL 2: Upload the CSV

Why: data access is the first integration step in any AI project.

```
from google.colab import files
uploaded = files.upload()
print("Uploaded:", list(uploaded.keys()))
```

CELL 3: Install optional libraries

Why: makes the notebook reproducible across teams.

```
!pip -q install mlxtend joblib
```

```
print("Installed: mlxtend, joblib")
```

CELL 4: Import libraries

Why: sets a consistent toolbox for the whole day.

```
import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.model_selection import train_test_split, cross_val_score, GridSearchCV
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.metrics import (
    mean_absolute_error, mean_squared_error, r2_score,
    accuracy_score, classification_report,
    confusion_matrix, ConfusionMatrixDisplay
)
from sklearn.linear_model import LinearRegression, LogisticRegression
from sklearn.ensemble import RandomForestRegressor, RandomForestClassifier
from sklearn.neural_network import MLPRegressor, MLPClassifier
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA

import joblib
```

CELL 5: Load the dataset

Why: confirms the file name and loads it into a DataFrame.

```
import os
print("Files in /content:", os.listdir("/content")[:30])

file_path = "/content/shopping_behavior_updated (1).csv"  # change if needed
df = pd.read_csv(file_path)

print("Rows, columns:", df.shape)
df.head()
```

==========

SECTION 2: EDA (EXPLORE BEFORE YOU MODEL)

==========

The goal of EDA is to answer:

- What is in the data?
- What looks wrong or risky?
- What patterns are visible?
- What decisions could this support?

CELL 6: Structure and data quality

Why: catches most issues early (types, missing values, duplicates).

```
print("Columns:")
for c in df.columns:
    print("-", c)
```

```
print("\nData types:")
print(df.dtypes)

print("\nMissing values:")
display(df.isnull().sum())

print("\nDuplicate rows:", int(df.duplicated().sum()))

print("\nNumeric summary:")
display(df.describe(numeric_only=True))
```

CELL 7: Categorical overview (top values per text column)

Why: reveals dominant segments, long tails, and messy labels.

```
cat_cols = df.select_dtypes(include="object").columns.tolist()
for col in cat_cols:
    print("\n---", col, "---")
    print("Unique:", df[col].nunique())
    display(df[col].value_counts().head(10))
```

Mini prompts to discuss

- Which categories dominate?
- Are there any labels that look inconsistent (spaces, spelling, casing)?

CELL 8: Numeric distributions (histograms and box plots)

Why: distributions and outliers change how you interpret averages and model results.

```
num_cols = df.select_dtypes(include=[np.number]).columns.tolist()


for col in num_cols:
    plt.figure(figsize=(7,4))
    plt.hist(df[col], bins=20, edgecolor="black")
    plt.title(f"Histogram: {col}")
    plt.xlabel(col)
    plt.ylabel("Count")
    plt.show()


    plt.figure(figsize=(7,2.5))
    sns.boxplot(x=df[col])
    plt.title(f"Box plot: {col}")
    plt.show()
```

CELL 9: Segment comparisons (turn data into decisions)

Why: this is where EDA becomes business insight.

```
def group_stats(group_col):
    out = df.groupby(group_col).agg(
        orders=("Customer ID", "count"),
        avg_spend=("Purchase Amount (USD)", "mean"),
        median_spend=("Purchase Amount (USD)", "median"),
        avg_rating=("Review Rating", "mean"),
        avg_prev_purchases=("Previous Purchases", "mean")
    ).sort_values("avg_spend", ascending=False)
    return out
```

```
for gc in ["Gender","Category","Season","Subscription Status","Discount Applied","Payment Method","Frequency of Purchases"]:
    print("\n====", gc, "====")
    display(group_stats(gc).head(15))
```

Interpretation prompts

- If discount is linked to higher spend, is it driving spend, or targeted at higher spenders?

- If subscribers spend more, is that an effect of subscription, or selection?

CELL 10: Pivot tables (two-way comparisons)

Why: exposes interactions (category by season, category by gender).

```
pivot_spend = pd.pivot_table(df, values="Purchase Amount (USD)", index="Category", columns="Season", aggfunc="mean")
display(pivot_spend)

pivot_count = pd.pivot_table(df, values="Customer ID", index="Category", columns="Gender", aggfunc="count")
display(pivot_count)
```

CELL 11: Correlations (numeric only)

Why: suggests relationships to explore. Caution: not causation.

```
num = df[["Age","Purchase Amount (USD)","Review Rating","Previous Purchases"]].copy()
corr = num.corr()

plt.figure(figsize=(6,5))
sns.heatmap(corr, annot=True, cmap="Blues")
plt.title("Correlation heatmap")
plt.show()
```

CELL 12: Outliers (IQR method) for purchase amount

Why: outliers can distort averages and models. Flag them, then decide what to do.

```
q1 = df["Purchase Amount (USD)"].quantile(0.25)
q3 = df["Purchase Amount (USD)"].quantile(0.75)
iqr = q3 - q1
lower = q1 - 1.5 * iqr
upper = q3 + 1.5 * iqr

outliers = df[(df["Purchase Amount (USD)"] < lower) | (df["Purchase Amount (USD)"] > upper)]
print("IQR bounds:", round(lower,2), "to", round(upper,2))
print("Outlier rows:", outliers.shape[0])
display(outliers.head(10))
```

CELL 13: Check repeat purchases (important for baskets and forecasting)

Why: association rules and forecasting need repeated events over time.

```
print("Unique customers:", df["Customer ID"].nunique(), "Rows:", df.shape[0])
counts = df["Customer ID"].value_counts()
print("Customers with 2+ rows:", int((counts >= 2).sum()))
print("Max rows for one customer:", int(counts.max()))
```

==========

SECTION 3: CLEANING AND FEATURE ENGINEERING

==========

CELL 14: Clean text columns and save a cleaned CSV

Why: small label issues break grouping and one hot encoding.

```
df_clean = df.copy()

for col in df_clean.select_dtypes(include="object").columns:
    df_clean[col] = df_clean[col].astype(str).str.strip()

clean_path = "/content/cleaned_shopping_data.csv"
df_clean.to_csv(clean_path, index=False)
print("Saved:", clean_path)
```

CELL 15: Simple engineered features

Why: better features often beat more complex models.

```
df_feat = df_clean.copy()
df_feat["subscriber_flag"] = (df_feat["Subscription Status"].str.lower() == "yes").astype(int)
df_feat["discount_flag"] = (df_feat["Discount Applied"].str.lower() == "yes").astype(int)

df_feat["spend_band"] = pd.cut(
    df_feat["Purchase Amount (USD)"],
    bins=[-np.inf, 40, 70, np.inf],
    labels=["low","medium","high"]
)
df_feat[["Purchase Amount (USD)","spend_band","subscriber_flag","discount_flag"]].head()
```

==========

SECTION 4: REGRESSION (PREDICT SPEND)

==========

Goal: predict Purchase Amount (USD)

Why this is useful

It supports value estimation (for planning, targeting, and prioritisation).

CELL 16: Preprocessing pipeline (recommended)

Why: prevents leakage and packages preprocessing with the model.

```
target_reg = "Purchase Amount (USD)"
X = df_feat.drop(columns=[target_reg])
y = df_feat[target_reg]

cat_features = X.select_dtypes(include="object").columns.tolist()
num_features = X.select_dtypes(include=[np.number]).columns.tolist()

preprocess = ColumnTransformer(
   transformers=[
      ("cat", OneHotEncoder(handle_unknown="ignore"), cat_features),
      ("num", StandardScaler(), num_features)
   ]
)
```

CELL 17: Train and compare three regressors (Linear, Random Forest, MLP)

Why: compare a simple baseline, a strong tabular model, and a neural baseline.

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

```
def eval_reg(name, y_true, y_pred):
    mae = mean_absolute_error(y_true, y_pred)
    rmse = np.sqrt(mean_squared_error(y_true, y_pred))
    r2 = r2_score(y_true, y_pred)
    print(name, "| MAE:", round(mae,2), "| RMSE:", round(rmse,2), "| R2:", round(r2,3))


reg_lr = Pipeline([("prep", preprocess), ("model", LinearRegression())])
reg_lr.fit(X_train, y_train)
eval_reg("Linear", y_test, reg_lr.predict(X_test))


reg_rf = Pipeline([("prep", preprocess), ("model", RandomForestRegressor(n_estimators=300, random_state=42))])
reg_rf.fit(X_train, y_train)
eval_reg("RandomForest", y_test, reg_rf.predict(X_test))


reg_mlp = Pipeline([("prep", preprocess), ("model", MLPRegressor(hidden_layer_sizes=(64,32), max_iter=400,
random_state=42))])
reg_mlp.fit(X_train, y_train)
eval_reg("MLP", y_test, reg_mlp.predict(X_test))
```

CELL 18: Residual plot (diagnose errors)

Why: shows whether the model systematically under or over predicts.

```
pred = reg_rf.predict(X_test)
residuals = y_test - pred

plt.figure(figsize=(7,4))
plt.scatter(pred, residuals, alpha=0.4)
plt.axhline(0, color="black")
plt.title("Residuals (Random Forest)")
```

```
plt.xlabel("Predicted spend")
plt.ylabel("Residual (actual - predicted)")
plt.show()
```

Optional: cross-validated R2

```
scores = cross_val_score(reg_rf, X, y, cv=5, scoring="r2")
print("CV R2 mean:", round(scores.mean(),3), "Std:", round(scores.std(),3))
```

==========

SECTION 5: CLASSIFICATION (PREDICT CATEGORY)

==========

Goal: predict Category

Why this is useful

It supports intent prediction, routing, and product mix planning.

CELL 19: Train baseline classifiers (Logistic, Random Forest, MLP)

Why: compare a simple baseline, a strong tabular baseline, and a neural baseline.

```
target_clf = "Category"
Xc = df_feat.drop(columns=[target_clf])
yc = df_feat[target_clf]

Xc_train, Xc_test, yc_train, yc_test = train_test_split(Xc, yc, test_size=0.2, random_state=42, stratify=yc)

cat_features_c = Xc.select_dtypes(include="object").columns.tolist()
num_features_c = Xc.select_dtypes(include=[np.number]).columns.tolist()
```

```
preprocess_c = ColumnTransformer(
    transformers=[
        ("cat", OneHotEncoder(handle_unknown="ignore"), cat_features_c),
        ("num", StandardScaler(), num_features_c)
    ]
)

clf_lr = Pipeline([("prep", preprocess_c), ("model", LogisticRegression(max_iter=1000))])
clf_lr.fit(Xc_train, yc_train)
pred_lr = clf_lr.predict(Xc_test)
print("Logistic accuracy:", round(accuracy_score(yc_test, pred_lr), 3))

clf_rf = Pipeline([("prep", preprocess_c), ("model", RandomForestClassifier(n_estimators=400, random_state=42))])
clf_rf.fit(Xc_train, yc_train)
pred_rf = clf_rf.predict(Xc_test)
print("RandomForest accuracy:", round(accuracy_score(yc_test, pred_rf), 3))

clf_mlp = Pipeline([("prep", preprocess_c), ("model", MLPClassifier(hidden_layer_sizes=(128,64), max_iter=400,
random_state=42))])
clf_mlp.fit(Xc_train, yc_train)
pred_mlp = clf_mlp.predict(Xc_test)
print("MLP accuracy:", round(accuracy_score(yc_test, pred_mlp), 3))

print("\nRandomForest classification report:")
print(classification_report(yc_test, pred_rf))
```

CELL 20: Confusion matrix and how to read it

Why: shows which categories are confused with which, and where to improve.

How to interpret

- diagonal = correct

- off diagonal = mistakes

Use it to decide whether to refine labels, add features, or rebalance classes.

```
labels = sorted(df_feat["Category"].unique().tolist())
cm = confusion_matrix(yc_test, pred_rf, labels=labels)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=labels)

plt.figure(figsize=(12,12))
disp.plot(xticks_rotation=90, values_format="d")
plt.title("Confusion matrix (Random Forest)")
plt.show()
```

CELL 21: Quick tuning example (GridSearchCV)

Why: demonstrates a structured improvement loop.

```
param_grid = {
    "model__n_estimators": [200, 400],
    "model__max_depth": [None, 10, 20],
    "model__min_samples_split": [2, 10],
}
tune = Pipeline([("prep", preprocess_c), ("model", RandomForestClassifier(random_state=42))])

grid = GridSearchCV(tune, param_grid=param_grid, cv=3, scoring="accuracy", n_jobs=-1)
grid.fit(Xc_train, yc_train)
```

```
print("Best params:", grid.best_params_)
print("Best CV accuracy:", round(grid.best_score_, 3))

best_model = grid.best_estimator_
pred_best = best_model.predict(Xc_test)
print("Test accuracy:", round(accuracy_score(yc_test, pred_best), 3))
```

Professional improvement checklist (use this on any dataset)

- label quality: clean mapping, merge overlapping classes if needed
- features: add meaningful signals, not just more columns
- evaluation: use macro F1 if classes are imbalanced
- imbalance: class_weight, resampling, collect more examples
- tuning: grid search, then narrower search
- error analysis: use confusion matrix and per-class recall

==========

SECTION 6: SEGMENTATION (CLUSTERING)

==========

Goal: group customers into segments and propose actions.

```
cluster_features = ["Age","Purchase Amount (USD)","Previous Purchases","Review Rating"]
X_cluster = df_feat[cluster_features].copy()

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_cluster)

k = 4
```

```
kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)


df_clusters = df_feat.copy()
df_clusters["Cluster"] = kmeans.fit_predict(X_scaled)


display(df_clusters["Cluster"].value_counts().sort_index())
display(df_clusters.groupby("Cluster")[cluster_features].mean())
```

Visualise clusters (optional)

```
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)


plt.figure(figsize=(7,5))
plt.scatter(X_pca[:,0], X_pca[:,1], c=df_clusters["Cluster"], alpha=0.6)
plt.title("Clusters (PCA projection)")
plt.xlabel("PCA 1")
plt.ylabel("PCA 2")
plt.show()
```

Mini tasks

- name each cluster based on its average profile
- write one business action per cluster

Save clustered data

```
clustered_path = "/content/clustered_customer_data.csv"
df_clusters.to_csv(clustered_path, index=False)
print("Saved:", clustered_path)
```

==========

SECTION 7: ASSOCIATION RULES (PRODUCT AFFINITY)

==========

Key requirement

Association rules need baskets or repeat purchases.

Run this check first:

```
counts = df_clean["Customer ID"].value_counts()
print("Customers with 2+ rows:", int((counts >= 2).sum()))
```

If it is near zero, you need to collect:

- order_id, order_date, customer_id, and multiple items per order

Small demo (teaches the technique even if the dataset is not suitable)

```
from mlxtend.frequent_patterns import apriori, association_rules

demo = pd.DataFrame([
    {"Bread":1, "Milk":1, "Eggs":0, "Butter":1},
    {"Bread":1, "Milk":1, "Eggs":1, "Butter":0},
    {"Bread":0, "Milk":1, "Eggs":1, "Butter":1},
    {"Bread":1, "Milk":0, "Eggs":0, "Butter":1},
    {"Bread":1, "Milk":1, "Eggs":0, "Butter":0},
])

freq = apriori(demo, min_support=0.4, use_colnames=True)
rules = association_rules(freq, metric="lift", min_threshold=1.0).sort_values("lift", ascending=False)
```

```
display(freq)

display(rules[["antecedents","consequents","support","confidence","lift"]].head(10))
```

==========

SECTION 8: FORECASTING NEXT MONTH ORDERS (AND LSTM)

==========

What you need

Forecasting needs time. To forecast next month orders by Category and Gender, collect:

- order_date (timestamp)

- category, gender

- order_id and quantity (useful)

- at least 12 to 24 months of history

Template for monthly aggregation (run only if you have an Order Date column)

```
# df_time = pd.read_csv(file_path)

# df_time["Order Date"] = pd.to_datetime(df_time["Order Date"])

# df_time["month"] = df_time["Order Date"].dt.to_period("M").dt.to_timestamp()

# monthly = df_time.groupby(["month","Category","Gender"]).size().reset_index(name="orders")

# display(monthly.head())
```

LSTM demo (synthetic time series, to learn the concept)

```
!pip -q install tensorflow

import tensorflow as tf

from tensorflow.keras import layers, models

from sklearn.metrics import mean_squared_error


np.random.seed(42)
```

```
months = 48
t = np.arange(months)
series = 100 + 10*np.sin(2*np.pi*t/12) + 0.5*t + np.random.normal(0, 3, size=months)
series = series.astype(np.float32)

def make_sequences(arr, window=12):
    Xs, ys = [], []
    for i in range(len(arr) - window):
        Xs.append(arr[i:i+window])
        ys.append(arr[i+window])
    Xs = np.array(Xs)[..., None]
    ys = np.array(ys)
    return Xs, ys

window = 12
X_seq, y_seq = make_sequences(series, window=window)
split = int(0.8 * len(X_seq))
X_train, X_test = X_seq[:split], X_seq[split:]
y_train, y_test = y_seq[:split], y_seq[split:]

model = models.Sequential([
    layers.Input(shape=(window, 1)),
    layers.LSTM(32),
    layers.Dense(1)
])
model.compile(optimizer="adam", loss="mse")
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=30, verbose=0)
```

```
pred = model.predict(X_test).flatten()
rmse = np.sqrt(mean_squared_error(y_test, pred))
print("Demo RMSE:", round(float(rmse), 2))

======================================================================
```

SECTION 9: EXPORT AND SHARE

```
======================================================================
```

Save an ML-ready encoded dataset

```
categorical_cols = df_clean.select_dtypes(include="object").columns.tolist()
df_encoded = pd.get_dummies(df_clean, columns=categorical_cols, drop_first=True)

encoded_path = "/content/encoded_shopping_data.csv"
df_encoded.to_csv(encoded_path, index=False)
print("Saved:", encoded_path)
```

Optional: save models for reuse

```
joblib.dump(reg_rf, "/content/regression_rf_pipeline.pkl")
joblib.dump(best_model, "/content/best_category_model.pkl")  # if you ran tuning
```

Download artefacts

```
from google.colab import files
files.download("/content/cleaned_shopping_data.csv")
files.download("/content/encoded_shopping_data.csv")
files.download("/content/clustered_customer_data.csv")
```

Interpreting outputs and results

How to read EDA charts

Histograms: look for the typical range, skew (long tail), multiple peaks (mixed groups), and extreme values. For spend, a right-skew is common, so compare both mean and median.

Box plots: the middle line is the median, the box is the middle 50 percent, and points outside the whiskers are outliers. Use box plots to compare categories fairly, not just by average.

Bar charts of counts: show what is frequent, but frequency is not value. Combine with average spend and margin to prioritise actions.

Scatter plots: look for patterns or trends. If dots are random, the relationship is weak. If errors grow with spend, consider segmenting high spenders or transforming the target.

How to read tables and pivots

Group summaries: compare orders, average spend, median spend, average rating, and previous purchases by segment. Use median spend when the distribution is skewed.

Pivot tables: show interactions such as category by season or category by gender. Use them to spot where a category performs strongly in one season or segment.

How to interpret correlation

Correlation ranges from -1 to +1. Values near 0 mean little linear relationship, but non-linear relationships can still exist. Correlation is a hypothesis generator, not proof of cause.

Regression results: MAE, RMSE, R2

MAE is the typical absolute error in dollars. RMSE penalises large errors more strongly. R2 shows how much of the variation the model explains.

Judge usefulness by comparing MAE or RMSE to typical order value. For example, an MAE of 5 on an average order of 60 is often useful, while an MAE of 25 may not be actionable.

Residual plot: points should scatter around zero. Curves suggest missing non-linear relationships, and funnel shapes suggest errors grow for high spend. Consider better features, segmentation, or log transforming spend.

Classification results: report and confusion matrix

Classification report: precision answers 'when we predict this category, how often are we correct?'. Recall answers 'of all true items in this category, how many did we find?'. F1 balances precision and recall.

Confusion matrix: rows are true categories and columns are predicted categories. Large diagonal values are correct predictions. Large off-diagonal values reveal which categories are being confused.

Use the confusion matrix to pick the top confusions and decide actions: add features to separate categories, collect more data for weaker categories, adjust class balance, or simplify the label set if the business does not need that granularity.

Consider a normalised confusion matrix (percentages) to compare categories fairly when class sizes differ. Accuracy can look good while minority categories perform poorly.

How to improve accuracy in practice

Feature improvements usually bring the biggest gains: add time, channel, campaign exposure, product price, basket size, recency, frequency, and customer tenure.

Handle imbalance: use class weights, resampling, or collect more samples for rare categories. Evaluate macro average f1-score when minority categories matter.

Tune models with cross-validation. Do error analysis by reviewing false positives and false negatives in the most important categories.

If the goal is business routing, define costs of errors and optimise for the right metric, not only accuracy.

Notes on forecasting and LSTM

Forecasting requires time. If you want next-month orders by category or gender, you need an order date or timestamp for each purchase.

Once you have dates, aggregate to monthly counts per category and gender, then apply a forecasting model. Start with simple baselines (seasonal averages), then consider ARIMA/Prophet, gradient boosting, or LSTM if you have enough history.

The LSTM section in the notebook demonstrates the method on a synthetic series to teach the workflow. Replace the synthetic series with real monthly counts once date data is available.