```
import pandas as pd
import numpy as np
from pathlib import Path
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, confusion_matrix, roc_auc_score, roc_curve, accuracy_score, f1_score
import matplotlib.pyplot as plt
import joblib
```

```
# ---------- Configuration ----------
FILE_PATH = "dataset3.xlsx"   # adjust path if needed
OUTPUT_MODEL_PATH = "feasibility_model.pkl"
TARGET = "is_feasible"
RANDOM_STATE = 42
TEST_SIZE = 0.20
# -----------------------------------
```

```
# 1) Load data
df = pd.read_excel(FILE_PATH, sheet_name=0)
print("Loaded dataset shape:", df.shape)

if TARGET not in df.columns:
    raise ValueError(f"Target column '{TARGET}' not found in dataset columns: {df.columns.tolist()}")

X = df.drop(columns=[TARGET])
y = df[TARGET]
```

```
Loaded dataset shape: (1000, 13)
```

```
# 2) identify numeric and categorical columns
numeric_cols = X.select_dtypes(include=["int64", "float64"]).columns.tolist()
categorical_cols = X.select_dtypes(include=["object", "category"]).columns.tolist()

print("Numeric columns:", numeric_cols)
print("Categorical columns:", categorical_cols)
```

```
Numeric columns: ['#', 'Feed Mol Frac A', 'Feed Mol Frac B', 'Feed Mol Frac C', 'Reflux Ratio', 'Entrainer Flow (mol/hr)', 'Ent
Categorical columns: ['System Type', 'Components', 'Entrainer Component', 'compA', 'compB']
```

```
# 3) preprocessing pipelines
numeric_transformer = Pipeline([
    ("imputer", SimpleImputer(strategy="median")),
    ("scaler", StandardScaler())
])

categorical_transformer = Pipeline([
    ("imputer", SimpleImputer(strategy="constant", fill_value="__MISSING__")),
    ("onehot", OneHotEncoder(handle_unknown="ignore"))
])

preprocessor = ColumnTransformer([
    ("num", numeric_transformer, numeric_cols),
    ("cat", categorical_transformer, categorical_cols)
])
```

```
# 4) train/test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=TEST_SIZE, stratify=y, random_state=RANDOM_STATE
)
print("Train size:", X_train.shape, "Test size:", X_test.shape)
```

```
Train size: (800, 12) Test size: (200, 12)
```

```
# 5) Pipelines for two models
pipe_lr = Pipeline([
    ("prep", preprocessor),
    ("clf", LogisticRegression(max_iter=2000, class_weight="balanced", random_state=RANDOM_STATE))
])
```

```
pipe_rf = Pipeline([
    ("prep", preprocessor),
    ("clf", RandomForestClassifier(n_estimators=200, class_weight="balanced", random_state=RANDOM_STATE))
])
```

```
# 6) Hyperparameter grids (small to keep runtime reasonable)
param_grid_lr = {
    "clf__C": [0.01, 0.1, 1, 10]
}

param_grid_rf = {
    "clf__n_estimators": [100, 200],
    "clf__max_depth": [5, 10, None],
    "clf__min_samples_split": [2, 5]
}
```

```
# 7) Grid search (RandomForest and LogisticRegression)
print("Starting GridSearchCV for RandomForest...")
gs_rf = GridSearchCV(pipe_rf, param_grid_rf, cv=5, scoring="f1", n_jobs=-1, verbose=1)
gs_rf.fit(X_train, y_train)
print("Best RF params:", gs_rf.best_params_)

print("Starting GridSearchCV for LogisticRegression...")
gs_lr = GridSearchCV(pipe_lr, param_grid_lr, cv=5, scoring="f1", n_jobs=-1, verbose=1)
gs_lr.fit(X_train, y_train)
print("Best LR params:", gs_lr.best_params_)

rf_best = gs_rf.best_estimator_
lr_best = gs_lr.best_estimator_
```

```
Starting GridSearchCV for RandomForest...
Fitting 5 folds for each of 12 candidates, totalling 60 fits
Best RF params: {'clf__max_depth': 5, 'clf__min_samples_split': 2, 'clf__n_estimators': 100}
Starting GridSearchCV for LogisticRegression...
Fitting 5 folds for each of 4 candidates, totalling 20 fits
Best LR params: {'clf__C': 0.1}
```

```
# 8) Evaluation helper
def evaluate_model(model, Xt, yt, name="Model"):
    y_pred = model.predict(Xt)
    try:
        y_proba = model.predict_proba(Xt)[:, 1]
    except:
        y_proba = None
    print(f"\n--- {name} ---")
    print("Accuracy:", accuracy_score(yt, y_pred))
    print(classification_report(yt, y_pred))
    if y_proba is not None:
        print("ROC AUC:", roc_auc_score(yt, y_proba))
    print("Confusion Matrix:\n", confusion_matrix(yt, y_pred))
    return y_proba

proba_rf = evaluate_model(rf_best, X_test, y_test, "RandomForest (best)")
proba_lr = evaluate_model(lr_best, X_test, y_test, "LogisticRegression (best)")
```

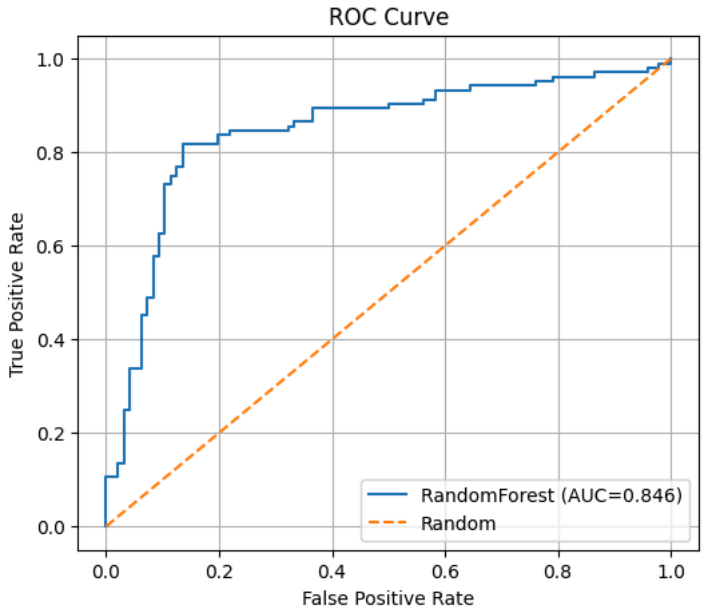
```
--- RandomForest (best) ---
Accuracy: 0.84
              precision    recall  f1-score   support

           0       0.81      0.86      0.84        96
           1       0.87      0.82      0.84       104

    accuracy                           0.84       200
   macro avg       0.84      0.84      0.84       200
weighted avg       0.84      0.84      0.84       200

ROC AUC: 0.846454326923077
Confusion Matrix:
 [[83 13]
 [19 85]]

--- LogisticRegression (best) ---
Accuracy: 0.835
              precision    recall  f1-score   support

           0       0.81      0.86      0.83        96
           1       0.87      0.81      0.84       104
```

```
    accuracy                           0.83       200
   macro avg       0.84      0.84      0.83       200
weighted avg       0.84      0.83      0.84       200

ROC AUC: 0.846454326923077
Confusion Matrix:
 [[83 13]
 [20 84]]
```

```
# 9) Choose best by F1 score on test set
f1_rf = f1_score(y_test, rf_best.predict(X_test))
f1_lr = f1_score(y_test, lr_best.predict(X_test))
print("\nF1 RF:", f1_rf, "F1 LR:", f1_lr)
if f1_rf >= f1_lr:
    best_model = rf_best
    best_name = "RandomForest"
else:
    best_model = lr_best
    best_name = "LogisticRegression"
print("Selected best model:", best_name)
```

```
F1 RF: 0.8415841584158416 F1 LR: 0.835820895522388
Selected best model: RandomForest
```

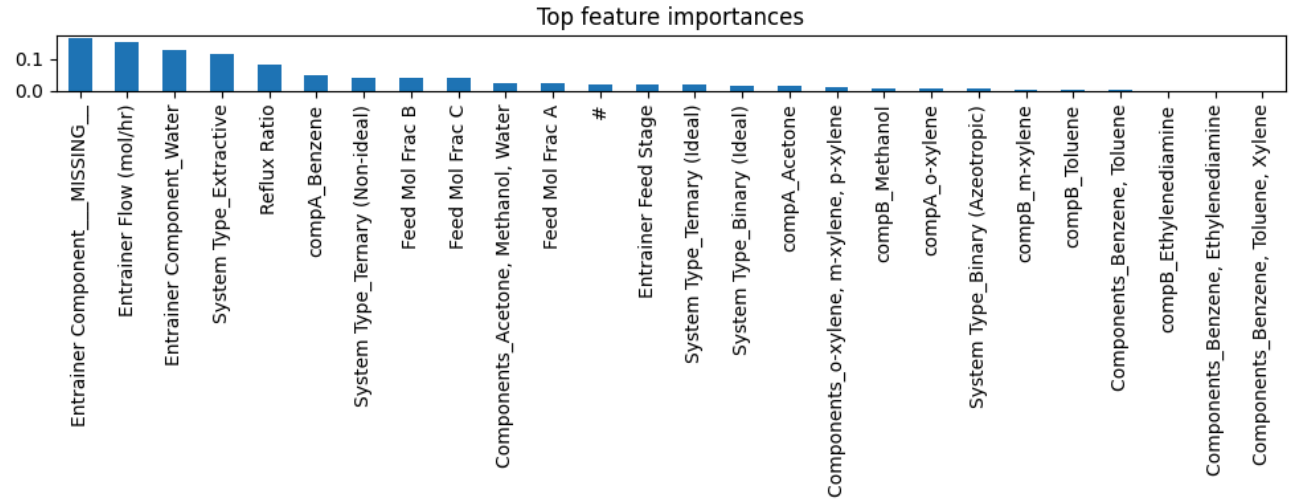
```
# 11) Plot ROC for the chosen model (if probabilities available)
try:
    y_proba_best = best_model.predict_proba(X_test)[:,1]
    fpr, tpr, _ = roc_curve(y_test, y_proba_best)
    plt.figure(figsize=(6,5))
    plt.plot(fpr, tpr, label=f"{best_name} (AUC={roc_auc_score(y_test, y_proba_best):.3f})")
    plt.plot([0,1],[0,1],"--", label="Random")
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate")
    plt.title("ROC Curve")
    plt.legend()
    plt.grid(True)
    plt.show()
except Exception as e:
    print("ROC unavailable:", e)
```




```
# 12) If RandomForest chosen, show feature importances
if best_name == "RandomForest":
    # Extract feature names after preprocessing
    ct = best_model.named_steps['prep']
    # numeric names
    num_names = numeric_cols
    # categorical onehot names:
    ohe = ct.named_transformers_['cat'].named_steps['onehot']
    try:
        cat_names = ohe.get_feature_names_out(categorical_cols).tolist()
    except:
        # fallback if older scikit-learn
        cat_names = []
        for i, cats in enumerate(ohe.categories_):
```

```
            col = categorical_cols[i]
            cat_names += [f"{col}__{c}" for c in cats]
    feat_names = num_names + cat_names
    importances = best_model.named_steps['clf'].feature_importances_
    fi = pd.Series(importances, index=feat_names).sort_values(ascending=False).head(30)
    print("\nTop feature importances:\n", fi)
    fi.plot(kind="bar", figsize=(10,4))
    plt.title("Top feature importances")
    plt.tight_layout()
    plt.show()

print("All done.")
```

```
Top feature importances:
 Entrainer Component___MISSING__           0.164571
Entrainer Flow (mol/hr)                    0.154597
Entrainer Component_Water                  0.128441
System Type_Extractive                     0.114762
Reflux Ratio                               0.081053
compA_Benzene                              0.047941
System Type_Ternary (Non-ideal)            0.041427
Feed Mol Frac B                            0.039897
Feed Mol Frac C                            0.039302
Components_Acetone, Methanol, Water        0.025285
Feed Mol Frac A                            0.022993
#                                          0.022176
Entrainer Feed Stage                       0.021757
System Type_Ternary (Ideal)                0.020864
System Type_Binary (Ideal)                 0.016139
compA_Acetone                              0.014445
Components_o-xylene, m-xylene, p-xylene    0.010557
compB_Methanol                             0.007266
compA_o-xylene                             0.006611
System Type_Binary (Azeotropic)            0.006322
compB_m-xylene                             0.003833
compB_Toluene                              0.003682
Components_Benzene, Toluene                0.002554
compB_Ethylenediamine                      0.001489
Components_Benzene, Ethylenediamine        0.001482
Components_Benzene, Toluene, Xylene        0.000556
dtype: float64
```




```
All done.
```

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d
from scipy.optimize import brentq
```

```
# ---------------------------
# Antoine constants (A, B, C) for Benzene & Toluene (example)
# units: log10(P_sat [mmHg]) = A - B / (T + C), T in °C
# Source: typical literature values (approx). Change as needed.
# ---------------------------
antoine = {
    "benzene": {"A": 6.90565, "B": 1211.033, "C": 220.79},
    "toluene": {"A": 6.95464, "B": 1344.8, "C": 219.48}
}

def psat_mmHg(antoine_params, T_C):
    A = antoine_params["A"]; B = antoine_params["B"]; C = antoine_params["C"]
    return 10 ** (A - B / (T_C + C))
```

```
def psat_Pa_from_mmHg(mmHg):
    # 1 mmHg = 133.322 Pa
    return mmHg * 133.322
```

```
# --------------------------
# Equilibrium: Raoult's Law for ideal binary
# y = pA / (pA + pB) where pA = x * P_A_sat, pB = (1-x) * P_B_sat
# (total pressure P cancels if both partial pressures divided by same P)
# More explicitly: y = x * P_A_sat / ( x*P_A_sat + (1-x)*P_B_sat )
# --------------------------
def equilibrium_curve(x_array, T_C, compA='benzene', compB='toluene'):
    psat_A = psat_Pa_from_mmHg(psat_mmHg(antoine[compA], T_C))
    psat_B = psat_Pa_from_mmHg(psat_mmHg(antoine[compB], T_C))
    # avoid division by zero
    denom = x_array * psat_A + (1 - x_array) * psat_B
    y = (x_array * psat_A) / denom
    return y
```

```
# --------------------------
# Operating lines
# rectifying: y = (R/(R+1)) x + xd/(R+1)
# q-line: y = q/(q-1) x - zf/(q-1)
# stripping: get line through (xb, xb) and intersection of q-line with rectifying line
# --------------------------
def rectifying_line(x, R, xd):
    return (R/(R+1.0)) * x + xd/(R+1.0)

def q_line(x, q, zf):
    # q != 1 handled; q=1 -> vertical line x = zf (we treat separately)
    return (q/(q-1.0)) * x - zf/(q-1.0)

def build_stripping_line_from_intersection(xb, xb_y, x_inter, y_inter):
    # line through (xb, xb_y) and (x_inter, y_inter)
    slope = (y_inter - xb_y) / (x_inter - xb) if x_inter != xb else 0.0
    def y_stripping(x):
        return slope * (x - xb) + xb_y
    return y_stripping, slope
```

```
# --------------------------
# Graphical stepping algorithm
# Steps between operating/eqn curves until x <= xb
# Input: xd (distillate y), xb (desired bottoms x), R, q, zf, T_C
# Returns: lists of stage coords for plotting and the feed stage index
# --------------------------
def mccabe_thiele_stepping(xd, xb, zf, R, q, T_C, compA='benzene', compB='toluene', P_atm=1.0, n_pts_eq=500):
    # equilibrium curve
    x_eq = np.linspace(0, 1, n_pts_eq)
    y_eq = equilibrium_curve(x_eq, T_C, compA, compB)
    eq_interp = interp1d(x_eq, y_eq, kind='cubic', bounds_error=False, fill_value='extrapolate')
    # Also build inverse mapping y->x by root finding on eq(x) - y = 0
    def x_from_y(y_target):
        # bracket search: find interval in x_eq where eq(x) crosses y_target
        # use brentq within small intervals
        # if monotonic-ish, we can try to invert by interpolation of x as function of y if monotonic
        # check monotonic: for typical binary, eq is monotonic increasing.
        # We'll use a robust method: interpolation of x vs y sorted by y.
        # Build once:
        return inv_interp(y_target)
    # attempt monotonic inverse via sorting
    sort_idx = np.argsort(y_eq)
    y_sorted = y_eq[sort_idx]
    x_sorted = x_eq[sort_idx]
    inv_interp = interp1d(y_sorted, x_sorted, kind='linear', bounds_error=False, fill_value=(0,1))
    # Operating lines
    def y_rect(x): return rectifying_line(x, R, xd)
    # find intersection between rectifying and q-line
    if abs(q - 1.0) < 1e-8:
        # q = 1 => feed is saturated liquid => q-line is vertical x = zf
        x_inter = zf
        y_inter = rectifying_line(x_inter, R, xd)
    else:
        # solve for x where rectifying_line(x) == q_line(x)
        # f(x) = rect - qline
        def f_inter(x):
            return rectifying_line(x, R, xd) - q_line(x, q, zf)
        # bracket root in [0,1]
        try:
```

```
            x_inter = brentq(f_inter, 0.0, 1.0)
            y_inter = rectifying_line(x_inter, R, xd)
        except Exception:
            # fallback: intersection at x=zf
            x_inter = zf
            y_inter = rectifying_line(x_inter, R, xd)
    # build stripping line using point (xb, xb) and (x_inter, y_inter)
    yb = xb
    y_strip_func, strip_slope = build_stripping_line_from_intersection(xb, yb, x_inter, y_inter)
    # stepping
    xs = [None]; ys = [xd]  # start at (x0,y0) where y0=xd, we'll first go horizontally to eq curve
    x_current = None
    y_current = xd
    stage_coords = []  # sequence of (x,y) points including horizontal & vertical moves
    stage_count = 0
    feed_stage_index = None
    max_iter = 2000
    for i in range(max_iter):
        # 1) horizontal to equilibrium: find x such that y_eq(x) = y_current
        # If y_current outside eq range, clamp
        if y_current <= np.min(y_eq):
            x_horiz = x_eq[np.argmin(y_eq)]
        elif y_current >= np.max(y_eq):
            x_horiz = x_eq[np.argmax(y_eq)]
        else:
            x_horiz = inv_interp(y_current)
        stage_coords.append((x_horiz, y_current))
```

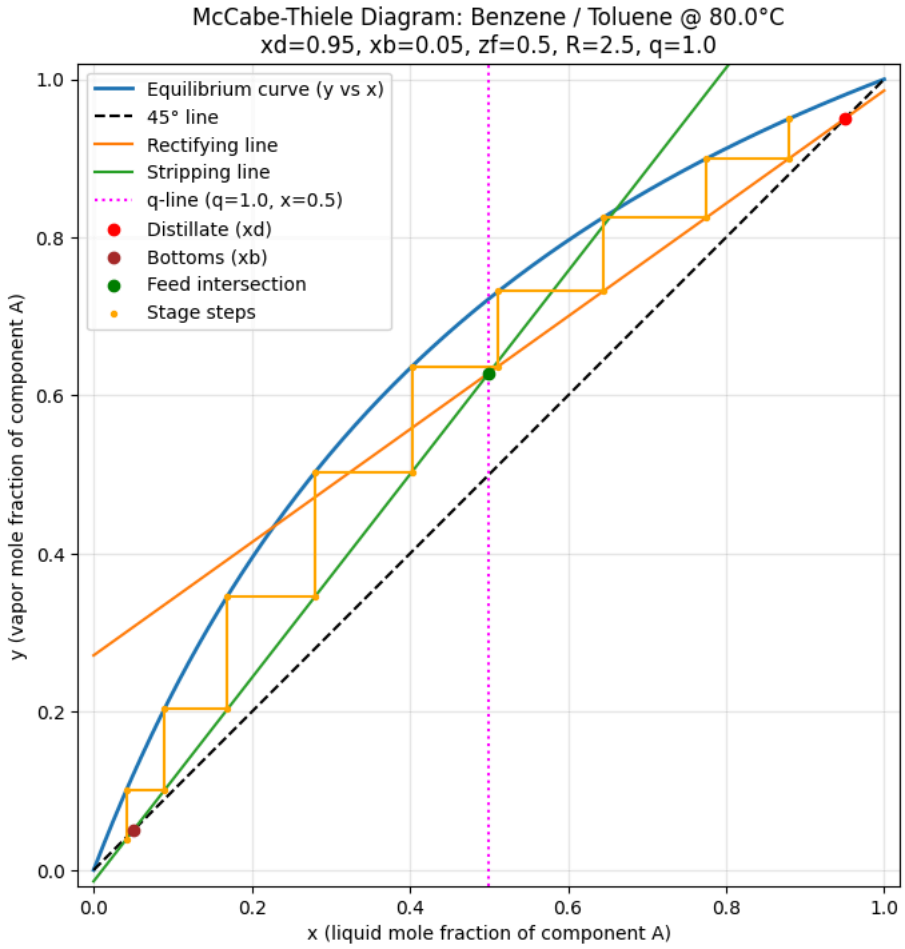
```
def mccabe_thiele_stepping(xd, xb, zf, R, q, T_C, compA='benzene', compB='toluene', n_pts_eq=500, max_iter=2000):
    """
    Perform McCabe-Thiele graphical stepping.
    Inputs:
        xd, xb : distillate and bottoms mole fractions (component A)
        zf : feed mole fraction (component A)
        R : reflux ratio
        q : feed quality (q-line)
        T_C : temperature in °C for using equilibrium_curve()
        compA, compB : component names (for equilibrium_curve)
        n_pts_eq : number of points to build eq curve
        max_iter : safety cap on number of stepping iterations
    Returns: dict with equilibrium arrays, operating line callables, stage coords, stair arrays,
             n_stages (int), feed_stage (int or None), intersection point
    """
    import numpy as np
    from scipy.interpolate import interp1d
    from scipy.optimize import brentq

    # Build equilibrium curve (x -> y)
    x_eq = np.linspace(0.0, 1.0, n_pts_eq)
    y_eq = equilibrium_curve(x_eq, T_C, compA, compB)  # assume this function exists
    # Create forward interpolator y(x)
    eq_interp = interp1d(x_eq, y_eq, kind='cubic', bounds_error=False, fill_value=(y_eq[0], y_eq[-1]))

    # Create inverse interpolator x(y). We sort by y before interpolation to avoid non-monotonic issues.
    sort_idx = np.argsort(y_eq)
    y_sorted = y_eq[sort_idx]
    x_sorted = x_eq[sort_idx]
    # ensure unique y values for interp (if duplicates exist, perturb slightly)
    # this protects interp1d from identical x for multiple y entries
    eps = 1e-12
    for i in range(1, len(y_sorted)):
        if y_sorted[i] <= y_sorted[i-1]:
            y_sorted[i] = y_sorted[i-1] + eps
    inv_interp = interp1d(y_sorted, x_sorted, kind='linear', bounds_error=False, fill_value=(0.0, 1.0))

    def x_from_y(y_target):
        """Return x such that eq_interp(x) ~= y_target using inverse interpolation."""
        # clamp y_target into [min(y_eq), max(y_eq)] to avoid extrapolation extremes
        y_min, y_max = y_sorted[0], y_sorted[-1]
        y_clamped = np.clip(y_target, y_min, y_max)
        return float(inv_interp(y_clamped))

    # Operating line definitions
    def y_rect(x):
        return (R / (R + 1.0)) * x + xd / (R + 1.0)

    def y_qline(x):
        # handle q==1 separately (vertical)
        return (q / (q - 1.0)) * x - zf / (q - 1.0)

    # Find intersection between rectifying line and q-line (or vertical if q==1)
    if abs(q - 1.0) < 1e-8:
```

```
        x_inter = float(zf)
        y_inter = float(y_rect(x_inter))
    else:
        def f_inter(x):
            return y_rect(x) - y_qline(x)
        # bracket root in [0,1] robustly
        try:
            x_inter = brentq(f_inter, 0.0, 1.0)
            y_inter = y_rect(x_inter)
        except Exception:
            # fallback: use zf as approximate intersection
            x_inter = float(zf)
            y_inter = float(y_rect(x_inter))

    # Build stripping line: line through (xb, xb) [bottoms point on diagonal] and intersection point
    x_b, y_b = float(xb), float(xb)  # (xb, xb)
    if abs(x_inter - x_b) < 1e-12:
        strip_slope = 0.0
    else:
        strip_slope = (y_inter - y_b) / (x_inter - x_b)

    def y_strip(x):
        return strip_slope * (x - x_b) + y_b

    # Now the graphical stepping:
    stage_coords = []   # will store alternating horizontal then vertical points: (x_horiz, y_current), (x_horiz, y_next)
    y_current = float(xd)   # start from vapor composition at top equal to xd (y = xd)
    stage_count = 0
    feed_stage_index = None

    for i in range(max_iter):
        # 1) horizontal to equilibrium: find x such that y_eq(x) = y_current
        x_horiz = x_from_y(y_current)
        # Append horizontal corner
        stage_coords.append((x_horiz, y_current))

        # 2) vertical to operating line (rectifying if x_horiz >= x_inter else stripping)
        if x_horiz >= x_inter:
            y_next = y_rect(x_horiz)
        else:
            y_next = y_strip(x_horiz)
        stage_coords.append((x_horiz, y_next))
        stage_count += 1

        # Record feed stage crossing (first time x goes below x_inter)
        if (x_horiz < x_inter) and (feed_stage_index is None):
            feed_stage_index = stage_count

        # Check stopping: if x_horiz <= xb (we reached bottoms composition)
        if x_horiz <= xb + 1e-8:
            break

        # Prepare next horizontal level from y_next
        y_current = y_next

        # safety
        if i == max_iter - 1:
            print("Warning: reached max iterations in stepping")

    # After loop: build stair arrays (x and y) for plotting steps.
    seg_x = []
    seg_y = []
    # stage_coords were appended as pairs: [(x1,y1),(x1,y2),(x2,y2),(x2,y3), ...]
    # We create stair segments by taking each pair
    if len(stage_coords) >= 2:
        # ensure even-length pairing
        pairs = list(zip(stage_coords[0::2], stage_coords[1::2]))
        for (x1, y1), (x2, y2) in pairs:
            # horizontal from (prev_x, prev_y) to (x1,y1) isn't necessary here because we store step corners
            # Append the horizontal corner then the vertical corner (both with same x)
            seg_x.extend([x1, x2])
            seg_y.extend([y1, y2])
    else:
        # no stages found, return empty stairs
        seg_x = []
        seg_y = []

    return {
        "x_eq": x_eq,
        "y_eq": y_eq,
        "rectifying": y_rect,
        "stripping": y_strip,
```

```
        "x_intersect": x_inter,
        "y_intersect": y_inter,
        "stage_coords": stage_coords,
        "stair_x": np.array(seg_x),
        "stair_y": np.array(seg_y),
        "n_stages": stage_count,
        "feed_stage": feed_stage_index
    }
```

```
# --------------------------
# Example usage / parameters
# --------------------------
if __name__ == "__main__":
    # User parameters
    T_C = 80.0          # Temperature in °C (choose appropriate for mixture)
    compA = 'benzene'
    compB = 'toluene'
    zf = 0.5            # feed mole fraction of component A (feed composition)
    xd = 0.95           # distillate composition (desired) in mole fraction of A (y_distillate)
    xb = 0.05           # bottoms composition desired (x_bottoms)
    R = 2.5             # reflux ratio
    q = 1.0             # feed condition: q=1 => saturated liquid, q=0 vapor, q>1 compressed liquid etc.

    # compute
    result = mccabe_thiele_stepping(xd=xd, xb=xb, zf=zf, R=R, q=q, T_C=T_C, compA=compA, compB=compB)

    print(f"Number of theoretical stages (including partial reboiler stage): {result['n_stages']}")
    print(f"Estimated feed-stage occurs at stage (counting from top): {result['feed_stage']}")
    print(f"Rectifying/stripping intersection (x,y): ({result['x_intersect']:.4f}, {result['y_intersect']:.4f})")

    # Plotting
    plt.figure(figsize=(8,8))
    # equilibrium curve
    plt.plot(result['x_eq'], result['y_eq'], label='Equilibrium curve (y vs x)', lw=2)
    # diagonal
    plt.plot([0,1],[0,1],'k--', label='45° line')
    # rectifying & stripping lines (plot over x range)
    x_plot = np.linspace(0,1,200)
    plt.plot(x_plot, result['rectifying'](x_plot), label='Rectifying line', lw=1.5)
    plt.plot(x_plot, result['stripping'](x_plot), label='Stripping line', lw=1.5)
    # q-line
    if abs(q - 1.0) < 1e-8:
        plt.axvline(zf, color='magenta', linestyle=':', label=f'q-line (q={q}, x={zf})')
    else:
        plt.plot(x_plot, q_line(x_plot, q, zf), color='magenta', linestyle=':', label=f'q-line (q={q})')
    # plot xy points for distillate & bottoms & feed intersection
    plt.scatter([xd], [xd], color='red', zorder=5, label='Distillate (xd)')
    plt.scatter([xb], [xb], color='brown', zorder=5, label='Bottoms (xb)')
    plt.scatter([result['x_intersect']], [result['y_intersect']], color='green', zorder=6, label='Feed intersection')

    # plot stepped stages
    stair_x = result['stair_x']
    stair_y = result['stair_y']
    if stair_x.size>0:
        # connect with lines
        # create sequence that starts at (xd, xd) => but our stair lists start from first horizontal point
        # For clarity, draw the step lines directly
        for i in range(0, len(stair_x)-1, 2):
            # horizontal
            plt.plot([stair_x[i], stair_x[i+1]], [stair_y[i], stair_y[i+1]], color='orange')
            # vertical
            if i+2 < len(stair_x):
                plt.plot([stair_x[i+1], stair_x[i+2]], [stair_y[i+1], stair_y[i+2]], color='orange')
        # also scatter stage corner points
        plt.scatter(stair_x, stair_y, s=8, color='orange', zorder=4, label='Stage steps')

    # Annotations
    plt.xlabel('x (liquid mole fraction of component A)')
    plt.ylabel('y (vapor mole fraction of component A)')
    plt.title(f"McCabe-Thiele Diagram: {compA.capitalize()} / {compB.capitalize()} @ {T_C}°C\nxd={xd}, xb={xb}, zf={zf}, R={R
    plt.xlim(-0.02,1.02)
    plt.ylim(-0.02,1.02)
    plt.legend(loc='best')
    plt.grid(alpha=0.3)
    plt.gca().set_aspect('equal', adjustable='box')
    plt.show()
```

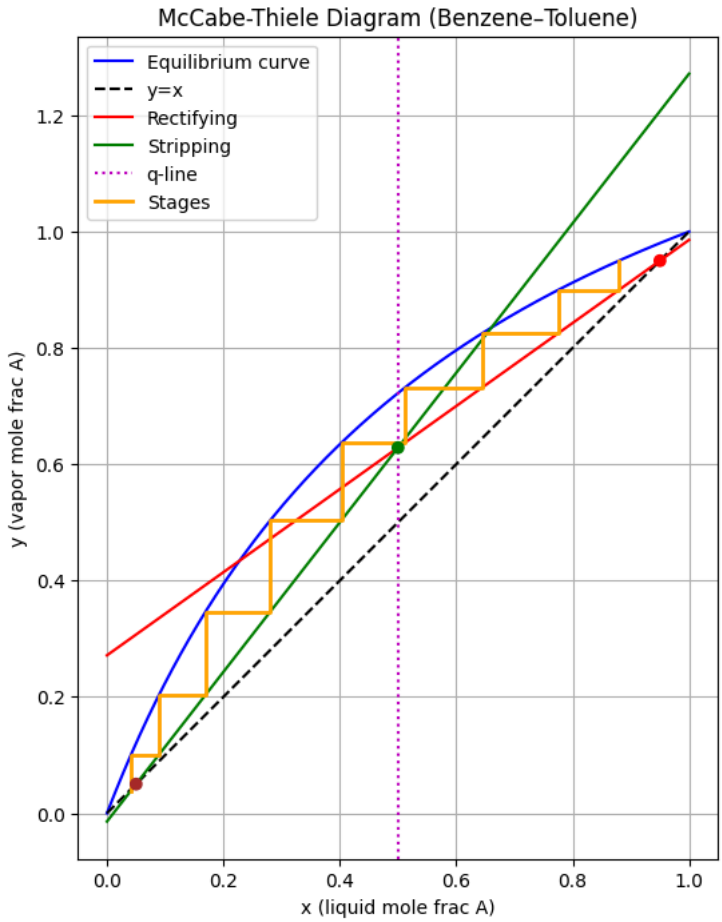
```
Number of theoretical stages (including partial reboiler stage): 9
Estimated feed-stage occurs at stage (counting from top): 5
Rectifying/stripping intersection (x,y): (0.5000, 0.6286)
```




```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d
from scipy.optimize import brentq

# ------------------------
# Antoine constants
# ------------------------
antoine = {
    "benzene": {"A": 6.90565, "B": 1211.033, "C": 220.79},
    "toluene": {"A": 6.95464, "B": 1344.8, "C": 219.48}
}

def psat_mmHg(antoine_params, T_C):
    A, B, C = antoine_params["A"], antoine_params["B"], antoine_params["C"]
    return 10 ** (A - B / (T_C + C))

def psat_Pa_from_mmHg(mmHg):
    return mmHg * 133.322

def equilibrium_curve(x_array, T_C, compA='benzene', compB='toluene'):
    psat_A = psat_Pa_from_mmHg(psat_mmHg(antoine[compA], T_C))
    psat_B = psat_Pa_from_mmHg(psat_mmHg(antoine[compB], T_C))
    denom = x_array * psat_A + (1 - x_array) * psat_B
    denom = np.where(denom == 0, 1e-12, denom)
    return (x_array * psat_A) / denom

# ------------------------
# McCabe–Thiele Stepping
# ------------------------
def mccabe_thiele_stepping(xd, xb, zf, R, q, T_C, compA='benzene', compB='toluene', n_pts_eq=500, max_iter=2000):
    x_eq = np.linspace(0, 1, n_pts_eq)
    y_eq = equilibrium_curve(x_eq, T_C, compA, compB)

    # Interpolators
    sort_idx = np.argsort(y_eq)
    y_sorted, x_sorted = y_eq[sort_idx], x_eq[sort_idx]
    inv_interp = interp1d(y_sorted, x_sorted, bounds_error=False, fill_value=(0,1))
    def x_from_y(y): return float(inv_interp(np.clip(y, y_sorted[0], y_sorted[-1])))
```

```
    # Operating lines
    y_rect = lambda x: (R/(R+1)) * x + xd/(R+1)
    if abs(q-1)<1e-8:
        x_inter, y_inter = zf, y_rect(zf)
    else:
        def f(x): return y_rect(x) - (q/(q-1))*x + zf/(q-1)
        x_inter = brentq(f, 0, 1); y_inter = y_rect(x_inter)
    slope_strip = (y_inter - xb)/(x_inter - xb)
    y_strip = lambda x: slope_strip*(x-xb)+xb

    # Stepping
    stage_coords = []
    y_current = xd
    stage_count, feed_stage = 0, None
    for i in range(max_iter):
        x_horiz = x_from_y(y_current)
        stage_coords.append((x_horiz,y_current))
        y_next = y_rect(x_horiz) if x_horiz>=x_inter else y_strip(x_horiz)
        stage_coords.append((x_horiz,y_next))
        stage_count+=1
        if x_horiz< x_inter and feed_stage is None: feed_stage=stage_count
        if x_horiz<=xb: break
        y_current=y_next

    seg_x, seg_y = [], []
    for (x1,y1),(x2,y2) in zip(stage_coords[0::2], stage_coords[1::2]):
        seg_x.extend([x1,x2]); seg_y.extend([y1,y2])

    return dict(x_eq=x_eq,y_eq=y_eq,rect=y_rect,strip=y_strip,
                x_int=x_inter,y_int=y_inter,
                stair_x=np.array(seg_x),stair_y=np.array(seg_y),
                n_stages=stage_count,feed_stage=feed_stage)

# ------------------------
# Example Run
# ------------------------
T_C=80; xd=0.95; xb=0.05; zf=0.5; R=2.5; q=1.0
res=mccabe_thiele_stepping(xd,xb,zf,R,q,T_C)

print(f"Stages: {res['n_stages']}, Feed stage: {res['feed_stage']}")
print(f"Intersection: ({res['x_int']:.3f}, {res['y_int']:.3f})")

# ------------------------
# Visualization
# ------------------------
plt.figure(figsize=(8,8))
plt.plot(res['x_eq'],res['y_eq'],'b',label='Equilibrium curve')
plt.plot([0,1],[0,1],'k--',label='y=x')
xx=np.linspace(0,1,200)
plt.plot(xx,res['rect'](xx),'r',label='Rectifying')
plt.plot(xx,res['strip'](xx),'g',label='Stripping')
if abs(q-1)<1e-8: plt.axvline(zf,c='m',ls=':',label='q-line')
else: plt.plot(xx,(q/(q-1))*xx-zf/(q-1),'m:',label='q-line')
plt.plot(res['stair_x'],res['stair_y'],'orange',lw=2,label='Stages')
plt.scatter([xd,xb,res['x_int']],[xd,xb,res['y_int']],c=['r','brown','green'],zorder=5)
plt.xlabel("x (liquid mole frac A)"); plt.ylabel("y (vapor mole frac A)")
plt.title("McCabe-Thiele Diagram (Benzene–Toluene)")
plt.legend(); plt.grid(True); plt.gca().set_aspect('equal')
plt.show()
```

```
Stages: 9, Feed stage: 5
Intersection: (0.500, 0.629)
```




```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, cross_val_score, StratifiedKFold
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import f1_score, accuracy_score
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer


# load
df = pd.read_excel("dataset_2000.xlsx", sheet_name=0)
TARGET = "is_feasible"

# 1) duplicates check
n_total = len(df)
n_unique = df.drop_duplicates().shape[0]
print("Total rows:", n_total, "Unique rows:", n_unique, "Duplicates:", n_total - n_unique)

# show how many duplicated rows appear in both train/test if you do a split
X = df.drop(columns=[TARGET])
y = df[TARGET]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=42)
# mark rows by their hash
# train_hashes = X_train.apply(lambda r: hash(tuple(r)), axis=1)
# test_hashes = X_test.apply(lambda r: hash(tuple(r)), axis=1)
# overlap = np.isin(test_hashes, train_hashes)
# print("Rows in test that are duplicates of rows in train:", overlap.sum())

# 2) cross-validation performance (on full dataset, deduplicated and original)
# Define preprocessing pipeline (reusing from previous cells)
numeric_cols = X.select_dtypes(include=["int64", "float64"]).columns.tolist()
categorical_cols = X.select_dtypes(include=["object", "category"]).columns.tolist()

preproc = ColumnTransformer([
    ("num", Pipeline([("impute", SimpleImputer(strategy="median")), ("scale", StandardScaler())]), numeric_cols),
    ("cat", Pipeline([("impute", SimpleImputer(strategy="constant", fill_value="__MISSING__")), ("ohe", OneHotEncoder(handle_
])

# Create a pipeline including preprocessing and the model
```

```
pipe_rf_cv = Pipeline([
    ("prep", preproc),
    ("clf", RandomForestClassifier(n_estimators=100, max_depth=None, random_state=42, class_weight="balanced"))
])


print("\nCross-val on ORIGINAL data (5-fold, f1):")
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
# Use the pipeline in cross_val_score
scores = cross_val_score(pipe_rf_cv, X, y, cv=cv, scoring="f1", n_jobs=-1)
print("F1 scores:", scores, "mean:", scores.mean())

# 3) deduplicate and drop suspicious columns (#)
clean = df.drop_duplicates().copy()
if '#' in clean.columns:
    clean = clean.drop(columns=['#'])
Xc = clean.drop(columns=[TARGET])
yc = clean[TARGET]

print("\nAfter dedup: rows:", len(clean))
print("Cross-val on DEDUP + drop('#'):")
# Define a new pipeline for the cleaned data if needed (or reuse preproc if columns are the same)
numeric_cols_c = Xc.select_dtypes(include=["int64", "float64"]).columns.tolist()
categorical_cols_c = Xc.select_dtypes(include=["object", "category"]).columns.tolist()

preproc_c = ColumnTransformer([
    ("num", Pipeline([("impute", SimpleImputer(strategy="median")), ("scale", StandardScaler())]), numeric_cols_c),
    ("cat", Pipeline([("impute", SimpleImputer(strategy="constant", fill_value="__MISSING__")), ("ohe", OneHotEncoder(handle_u
])

pipe_rf_cv_c = Pipeline([
    ("prep", preproc_c),
    ("clf", RandomForestClassifier(n_estimators=100, max_depth=None, random_state=42, class_weight="balanced"))
])

# Use the pipeline with cleaned data in cross_val_score
scores2 = cross_val_score(pipe_rf_cv_c, Xc, yc, cv=cv, scoring="f1", n_jobs=-1)
print("F1 scores:", scores2, "mean:", scores2.mean())

# 4) quick train/test after dedup to compare
# Reuse the pipeline defined for cleaned data (pipe_rf_cv_c)
X_train, X_test, y_train, y_test = train_test_split(Xc, yc, test_size=0.2, stratify=yc, random_state=42)
pipe_rf_cv_c.fit(X_train, y_train)
y_pred = pipe_rf_cv_c.predict(X_test)
print("Post-clean test accuracy:", accuracy_score(y_test, y_pred), "F1:", f1_score(y_test, y_pred))
```

```
Total rows: 2000 Unique rows: 60 Duplicates: 1940

Cross-val on ORIGINAL data (5-fold, f1):
F1 scores: [1. 1. 1. 1. 1.] mean: 1.0

After dedup: rows: 60
Cross-val on DEDUP + drop('#'):
F1 scores: [0.57142857 0.53333333 0.57142857 0.2        0.46153846] mean: 0.4675545787545787
```