KNN for Loan Default Predictions

Install xlrd for Excel support and import the core analysis, plotting, and k-NN libraries.

In [18]:
```
%pip install xlrd
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

from sklearn.neighbors import KNeighborsClassifier
from sklearn.cluster import KMeans
```

```
Requirement already satisfied: xlrd in c:\users\betoq\onedrive\documentos\github\exc
hange-data-concepts\.venv\lib\site-packages (2.0.2)
Note: you may need to restart the kernel to use updated packages.
```

Load the credit default dataset from Excel, rename the target column to TARGET, and preview the data.

In [19]:
```
DATA_FILE = "default of credit card clients.xls"

raw_df = (
    pd.read_excel(DATA_FILE, header=1)
    .rename(columns={"default payment next month": "TARGET"})
)

raw_df.head()
```

Out[19]:

	ID	LIMIT_BAL	SEX	EDUCATION	MARRIAGE	AGE	PAY_0	PAY_2	PAY_3	PAY_4	...	E
0	1	20000	2	2	1	24	2	2	-1	-1	...	
1	2	120000	2	2	2	26	-1	2	0	0	...	
2	3	90000	2	2	2	34	0	0	0	0	...	
3	4	50000	2	2	1	37	0	0	0	0	...	
4	5	50000	1	2	1	57	-1	0	-1	0	...	

5 rows × 25 columns

1.1 Prepare Data

Split the dataframe into feature columns (X) and the TARGET labels (y), then display both.

In [20]:
```
X = raw_df.drop(columns = 'TARGET')

y = raw_df['TARGET']
```

```
display(X,y)
```

	ID	LIMIT_BAL	SEX	EDUCATION	MARRIAGE	AGE	PAY_0	PAY_2	PAY_3	PAY_4
0	1	20000	2	2	1	24	2	2	-1	-1
1	2	120000	2	2	2	26	-1	2	0	0
2	3	90000	2	2	2	34	0	0	0	0
3	4	50000	2	2	1	37	0	0	0	0
4	5	50000	1	2	1	57	-1	0	-1	0
...	...	...	...	...	...	...	...	...	...	...
29995	29996	220000	1	3	1	39	0	0	0	0
29996	29997	150000	1	3	2	43	-1	-1	-1	-1
29997	29998	30000	1	2	2	37	4	3	2	-1
29998	29999	80000	1	3	1	41	1	-1	0	0
29999	30000	50000	1	2	1	46	0	0	0	0

30000 rows × 24 columns

```
0        1
1        1
2        0
3        0
4        0
       ..
29995    0
29996    0
29997    1
29998    1
29999    1
Name: TARGET, Length: 30000, dtype: int64
```

1.2(a) Split data

Create an 80/20 train-test split with a fixed seed and then carve out a validation fold from the training data for threshold tuning.

In [21]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.2,
    random_state=0,
    stratify=y,
)  # 20% of data is test here
```

```
X_train_sub, X_val, y_train_sub, y_val = train_test_split(
    X_train,
    y_train,
    test_size=0.25,
    random_state=0,
    stratify=y_train,
)  # carve out a validation fold (20% of original data) for threshold tuning

display(X_train_sub.shape, X_val.shape, X_test.shape, y_train_sub.shape, y_val.shap
```

```
(18000, 24)
(6000, 24)
(6000, 24)
(18000,)
(6000,)
(6000,)
```

1.2(b) Scale features

Standardize features using training-fold statistics and confirm the means and standard deviations of the scaled sets.

In [22]:
```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()                          # define the scaler

scaler.fit(X_train_sub)                            # train the scaler on training
X_train_scaled = scaler.transform(X_train_sub)     # apply the scaler to transform
X_val_scaled = scaler.transform(X_val)             # transform validation data
X_test_scaled = scaler.transform(X_test)           # apply the scaler to transform

dataframe_scaled = pd.DataFrame(np.round(X_train_scaled, 2), columns=X.columns)
dataframe_scaled
dataframe_scaled.describe()
```

Out[22]:

	ID	LIMIT_BAL	SEX	EDUCATION	MARRIAGE	AGE
count	18000.000000	18000.000000	18000.000000	18000.000000	18000.000000	18000.000000
mean	0.000002	0.000248	0.001183	0.000312	-0.002304	0.000369
std	1.000046	1.000078	0.999761	0.998659	0.998929	1.000128
min	-1.740000	-1.220000	-1.250000	-2.350000	-2.980000	-1.580000
25%	-0.860000	-0.910000	-1.250000	-1.080000	-1.060000	-0.810000
50%	0.010000	-0.220000	0.800000	0.180000	0.860000	-0.160000
75%	0.860000	0.560000	0.800000	0.180000	0.860000	0.610000
max	1.740000	4.880000	0.800000	5.250000	2.790000	4.320000

8 rows × 24 columns

1.3 Modeling

1.3(a) find best K

Set up a k-NN classifier and run a 5-fold grid search over odd k values to find the best parameter.

In [23]:
```
from sklearn.model_selection import GridSearchCV

knn = KNeighborsClassifier()

k_range = {'n_neighbors': np.arange(1,80,2)}                              #  k = 1,3,

grid = GridSearchCV(estimator = knn, param_grid = k_range, cv = 5)      # 5-CV on t

grid.fit(X_train_scaled, y_train_sub)       # search over the values on Train data

best_param = grid.best_params_          # k value that returns highest mean cv scor
best_cv_score = grid.best_score_        # mean cv score of the best k

print("Best Params: {}".format(best_param))
print("Mean cv score of the best k: {:.2%}".format(best_cv_score))
```

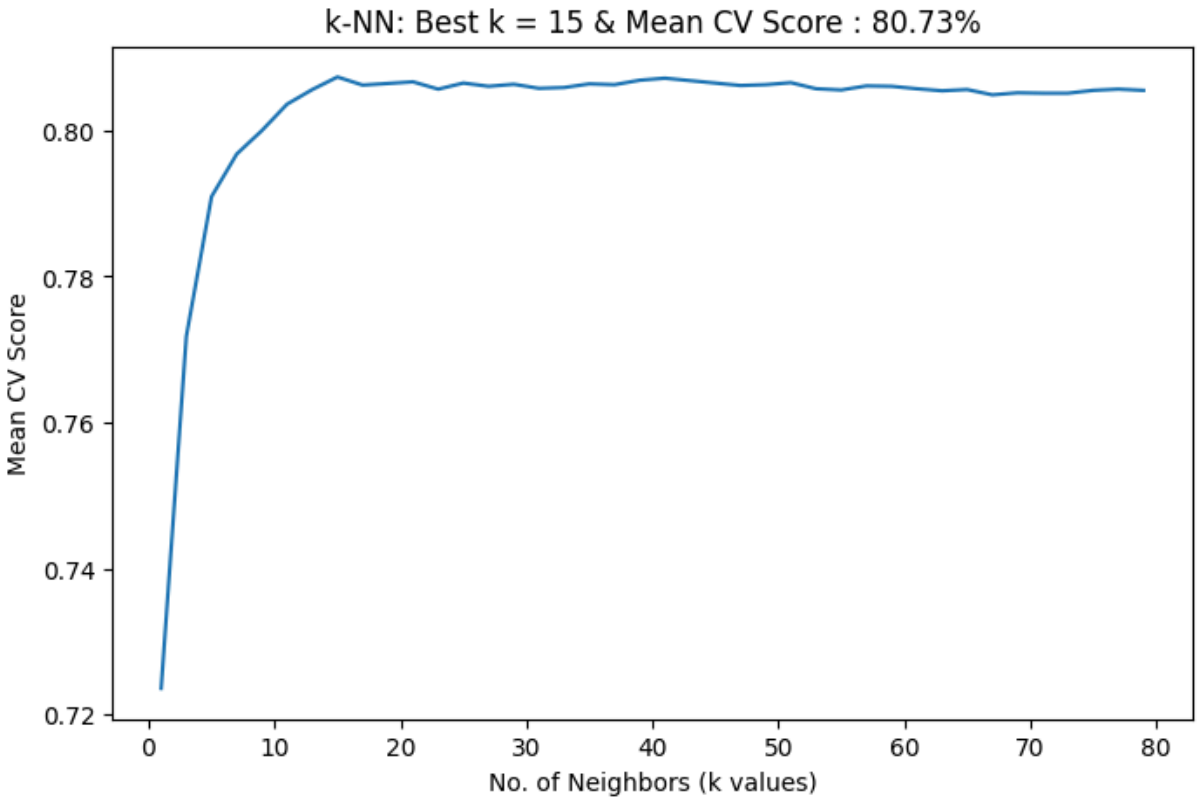
```
Best Params: {'n_neighbors': np.int64(15)}
Mean cv score of the best k: 80.73%
```

1.3(b) Visualize mean cv scores for each k value

Plot mean cross-validation accuracy versus k to visualize how performance changes with neighborhood size.

In [24]:
```
k_values = grid.cv_results_['param_n_neighbors']        # k values, same as k_range
cv_scores = grid.cv_results_['mean_test_score']         # mean cv scores for each k

plt.figure(figsize = (8, 5))
plt.plot(k_values,  cv_scores)        # a line plot (default blue solid line)
plt.xlabel('No. of Neighbors (k values)')
plt.ylabel('Mean CV Score')
plt.title('k-NN: Best k = {} & Mean CV Score : {:.2%}'.format(best_param['n_neighbo
plt.show()
```



Compute average credit limits for defaulters and non-defaulters to ground the cost assumptions.

In [25]:
```
# 1. Separate defaulters vs non-defaulters
df_default    = raw_df[raw_df["TARGET"] == 1]  # will default
df_no_default = raw_df[raw_df["TARGET"] == 0]  # will NOT default

# 2. Typical credit limit (principal proxy)
mean_limit_default    = df_default["LIMIT_BAL"].mean()
mean_limit_no_default = df_no_default["LIMIT_BAL"].mean()

print("Avg credit limit (default):    ", mean_limit_default)
print("Avg credit limit (no default):", mean_limit_no_default)
```

```
Avg credit limit (default):     130109.65641952984
Avg credit limit (no default): 178099.72607430234
```

Apply the PDF's six-month observation window and APR assumptions to derive per-period profit/loss inputs for the cost matrix.

In [26]:
```
# Business assumptions grounded in the PDF's six-month observation window
# April-September 2005 appears in the data dictionary PDF, so model six monthly bil
assumption_config = {
    "annual_apr": 0.18,             # 18% annual percentage rate from the reference
    "periods_per_year": 12,         # monthly compounding
    "observation_months": 6,        # six billing cycles in the PDF window (Apr-Sep
    "loss_given_default": 0.5,      # lose 50% of the limit if they default
}

annual_apr = assumption_config["annual_apr"]
periods_per_year = assumption_config["periods_per_year"]
observation_months = assumption_config["observation_months"]
loss_given_default = assumption_config["loss_given_default"]

periodic_rate = annual_apr / periods_per_year
period_length_months = 12 / periods_per_year
periods_in_window = observation_months / period_length_months

# Approx profit from approving a good customer (non-defaulter) over the observation
profit_good = mean_limit_no_default * ((1 + periodic_rate) ** periods_in_window - 1

# Approx loss from approving a bad customer (defaulter)
loss_bad = mean_limit_default * loss_given_default

print("Periodic rate: {:.2%}".format(periodic_rate))
print("Approx profit per good customer: {:.2f}".format(profit_good))
print("Approx loss per bad customer:    {:.2f}".format(loss_bad))
```

```
Periodic rate: 1.50%
Approx profit per good customer: 16642.22
Approx loss per bad customer:    65054.83
```

Construct a cost/benefit matrix that maps actual/predicted outcomes to monetary values using the assumptions.

In [27]:
```
# 3. Cost/benefit matrix driven by the data

# Story:
# TARGET=1 = will default
# Prediction=1 = treat as risky (reject / restrict credit)
# Prediction=0 = treat as safe (approve / keep credit)

value_TN =  profit_good   # Actual 0, Pred 0: good customer, approved → earn intere
value_FP =  0.0           # Actual 0, Pred 1: good customer, rejected → lose that p
value_FN = -loss_bad      # Actual 1, Pred 0: bad customer, approved → lose money
value_TP =  0.0           # Actual 1, Pred 1: bad customer, rejected → avoided loss

value_matrix = pd.DataFrame(
    {
        0: {0: value_TN, 1: value_FN},  # Predicted 0
        1: {0: value_FP, 1: value_TP},  # Predicted 1
    }
)
value_matrix.index.name = "Actual"
value_matrix.columns.name = "Predicted"
```

```
value_matrix
```

Out[27]:

Predicted	0	1
Actual		
0	16642.219712	0.0
1	-65054.828210	0.0

Define a helper that summarizes classification metrics, rates, AUCs, and optional utility for a model.

In [28]:

```
from sklearn.metrics import (
    average_precision_score,
    confusion_matrix,
    f1_score,
    precision_score,
    recall_score,
    roc_auc_score,
)

import numpy as np
import pandas as pd


def summarize_model(
    model_name,
    dataset_name,
    y_true,
    y_pred,
    y_score,
    threshold,
    utility_fn=None,
):
    '''Return a one-row DataFrame with key classification metrics.'''
    y_true_arr = np.asarray(y_true).ravel()
    y_pred_arr = np.asarray(y_pred).ravel()

    score_arr = None
    if y_score is not None:
        score_arr = np.asarray(y_score)
        if score_arr.ndim > 1:
            score_arr = score_arr[:, -1]
        score_arr = score_arr.ravel()

    tn, fp, fn, tp = confusion_matrix(
        y_true_arr, y_pred_arr, labels=[0, 1]
    ).ravel()
    total = tn + fp + fn + tp

    def _safe_div(num, denom):
        return num / denom if denom else np.nan
```

```
    fprate = _safe_div(fp, fp + tn)
    fnrate = _safe_div(fn, fn + tp)
    approval_rate = _safe_div(tn + fn, total)
    rejection_rate = _safe_div(fp + tp, total)

    roc_auc = np.nan
    pr_auc = np.nan
    if score_arr is not None and len(np.unique(y_true_arr)) > 1:
        try:
            roc_auc = roc_auc_score(y_true_arr, score_arr)
        except ValueError:
            pass
        try:
            pr_auc = average_precision_score(y_true_arr, score_arr)
        except ValueError:
            pass

    precision = precision_score(y_true_arr, y_pred_arr, zero_division=0)
    recall = recall_score(y_true_arr, y_pred_arr, zero_division=0)
    f1 = f1_score(y_true_arr, y_pred_arr, zero_division=0)

    utility_total = np.nan
    utility_per_app = np.nan
    if utility_fn is not None:
        utility_result = utility_fn(tn, fp, fn, tp)
        if isinstance(utility_result, dict):
            utility_total = utility_result.get('utility_total', np.nan)
            utility_per_app = utility_result.get('utility_per_app', np.nan)
        elif isinstance(utility_result, (tuple, list)):
            if len(utility_result) >= 2:
                utility_total, utility_per_app = utility_result[:2]
            elif len(utility_result) == 1:
                utility_total = utility_result[0]
        else:
            utility_total = utility_result

        try:
            needs_per_app = np.isnan(utility_per_app)
        except TypeError:
            needs_per_app = False

        if needs_per_app:
            utility_per_app = _safe_div(utility_total, total)

    summary = {
        'model_name': model_name,
        'dataset_name': dataset_name,
        'threshold': threshold,
        'TN': tn,
        'FP': fp,
        'FN': fn,
        'TP': tp,
        'fprate': fprate,
        'fnrate': fnrate,
        'approval_rate': approval_rate,
        'rejection_rate': rejection_rate,
```

```
        'roc_auc': roc_auc,
        'pr_auc': pr_auc,
        'precision': precision,
        'recall': recall,
        'f1': f1,
        'utility_total': utility_total,
        'utility_per_app': utility_per_app,
    }

    return pd.DataFrame([summary])
```

Find the probability threshold that maximizes expected utility using validation data and summarize the tuned model on the test data.

In [29]:
```
# Summarize the tuned k-NN model using a validation-tuned threshold on the held-out

def _knn_utility(tn, fp, fn, tp):
    total = (tn * value_TN) + (fp * value_FP) + (fn * value_FN) + (tp * value_TP)
    total_apps = tn + fp + fn + tp
    per_app = total / total_apps if total_apps else np.nan
    return {
        'utility_total': total,
        'utility_per_app': per_app,
    }


y_score_val = grid.predict_proba(X_val_scaled)[:, 1]
_knn_model_name = "k-NN (k={})".format(grid.best_params_['n_neighbors'])

threshold_grid = np.linspace(0.0, 1.0, 101)
threshold_summaries = []
for _thr in threshold_grid:
    y_pred_thr = (y_score_val >= _thr).astype(int)
    threshold_summaries.append(
        summarize_model(
            model_name=_knn_model_name,
            dataset_name="Validation",
            y_true=y_val,
            y_pred=y_pred_thr,
            y_score=y_score_val,
            threshold=_thr,
            utility_fn=_knn_utility,
        )
    )

threshold_results = pd.concat(threshold_summaries, ignore_index=True)

best_idx = threshold_results['utility_per_app'].fillna(-np.inf).idxmax()
best_threshold = threshold_results.loc[best_idx, 'threshold']

y_score_test = grid.predict_proba(X_test_scaled)[:, 1]
y_pred_best = (y_score_test >= best_threshold).astype(int)
knn_summary_df = summarize_model(
    model_name=_knn_model_name,
    dataset_name="Test",
```

```
    y_true=y_test,
    y_pred=y_pred_best,
    y_score=y_score_test,
    threshold=best_threshold,
    utility_fn=_knn_utility,
)

print(best_threshold)

knn_summary_df
```

```
0.21
```

Out[29]:

	model_name	dataset_name	threshold	TN	FP	FN	TP	fprate	fnrate	appr
0	k-NN (k=15)	Test	0.21	3529	1144	553	774	0.244811	0.416729	

1.3(c) Evaluate and apply the best model (validation-tuned)

Use the validation-tuned threshold to generate test-set predictions and view the label counts.

In [30]:
```
y_pred_test = y_pred_best

display(y_test, y_pred_test)
```

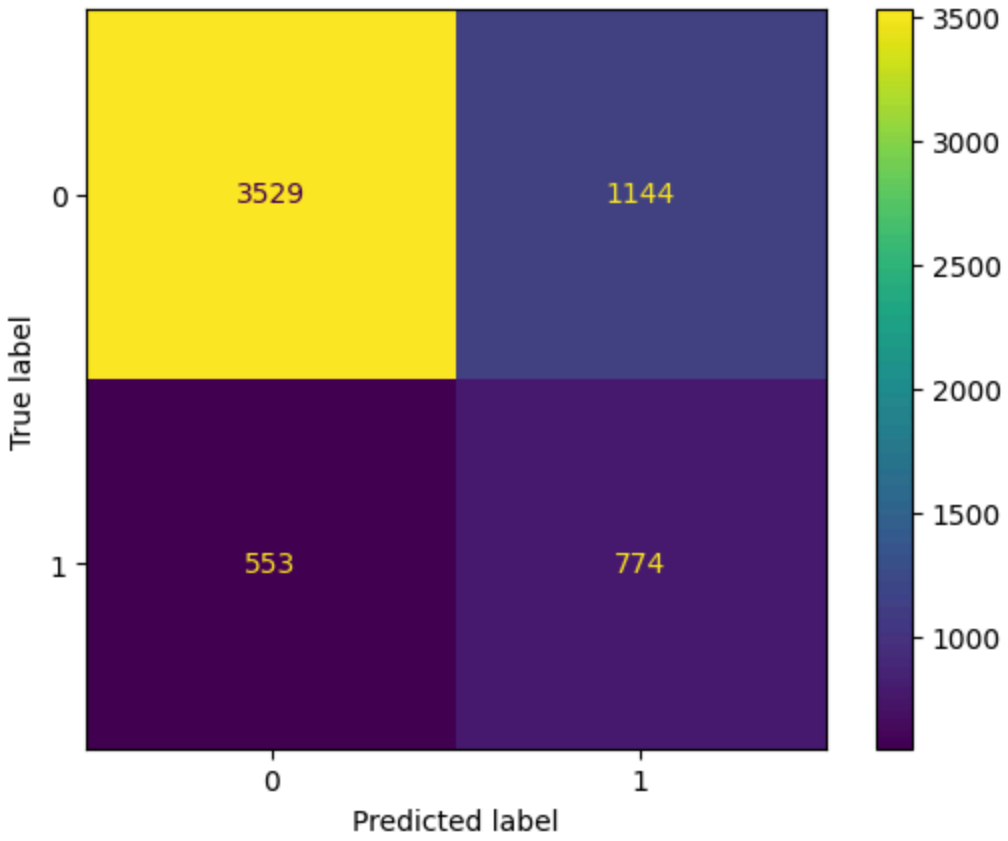
```
20553    0
6695     0
28997    0
2232     0
4165     1
        ..
25641    0
16907    1
1236     1
3013     0
16808    0
Name: TARGET, Length: 6000, dtype: int64
array([0, 1, 0, ..., 1, 0, 0], shape=(6000,))
```

2. Model Evaluation

Confusion matrix based on the tuned threshold.

In [31]:
```
from sklearn.metrics import ConfusionMatrixDisplay

disp = ConfusionMatrixDisplay.from_predictions(y_test, y_pred_test, display_labels=
```



Tabulate the raw confusion matrix counts for the test set using the tuned cutoff.

In [32]:
```
cm_counts = pd.crosstab(
    y_test,
    y_pred_test,
    rownames=['Actual'],
    colnames=['Predicted'],
    dropna=False
)

cm_counts
```

Out[32]:

Actual \ Predicted	0	1
0	3529	1144
1	553	774

Convert confusion matrix counts into probabilities for expected-value calculations driven by the tuned model.

In [33]:
```
# 2.2 Convert confusion matrix counts to probabilities (expected rates)

N = cm_counts.to_numpy().sum()      # total number of test samples
cm_prob = cm_counts / N             # element-wise division
```

```
cm_prob
```

Out[33]:

Predicted	0	1
Actual		
0	0.588167	0.190667
1	0.092167	0.129000

Calculate expected value per applicant and for the full test set using the tuned confusion matrix.

In [34]:
```
# 2.4 Expected value per person and in total

# EV per person = sum over all cells of [probability * value]
ev_per_person = (cm_prob * value_matrix).to_numpy().sum()

# EV over all people in the test set (just multiply by N)
ev_total = ev_per_person * N

print("Expected value per person: {:.2f}".format(ev_per_person))
print("Expected value over all {} test customers: {:.2f}".format(N, ev_total))
```

```
Expected value per person: 3792.51
Expected value over all 6000 test customers: 22755073.36
```