MMA/MMAB 869: Individual Assignment

- Connor Sullivan
- 20504818
- MMA 2026W
- 08/13/2025

Question 1: Uncle Steve's Diamonds

1.0: Load data

```
import pandas as pd
import numpy as np

# DO NOT MODIFY THIS CELL
df1 = pd.read_csv("https://drive.google.com/uc?export=download&id=1thHDCwQK3GijytoSS
df1.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 505 entries, 0 to 504
Data columns (total 4 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   Age            505 non-null    int64  
 1   Income         505 non-null    int64  
 2   SpendingScore  505 non-null    float64
 3   Savings        505 non-null    float64
dtypes: float64(2), int64(2)
memory usage: 15.9 KB
```

1.1: Clustering Algorithm #1

```
import time
import pandas as pd
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import silhouette_score, calinski_harabasz_score

# 1. Feature scaling (important for KMeans)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(df1)
```

```
# 2. Create and fit KMeans
best_silhouette = -1
best_calinski = -1
best_k = None
results = []

for n_clusters in range(2, 8):
    start_kmeans = time.time()
    kmeans = KMeans(n_clusters=n_clusters, random_state=42, n_init=10)
    labels = kmeans.fit_predict(X_scaled)
    end_kmeans = time.time()
    sil = silhouette_score(X_scaled, labels)
    cal = calinski_harabasz_score(X_scaled, labels)
    results.append({
        "n_clusters": n_clusters,
        "silhouette": sil,
        "calinski": cal,
        "time": end_kmeans - start_kmeans
    })
    print(
        f"KMeans (k={n_clusters}): silhouette={sil:.3f}, calinski={cal:.2f}, time={e
    )
    if sil > best_silhouette:
        best_silhouette = sil
        best_calinski = cal
        best_k = n_clusters
        best_labels = labels
        best_kmeans = kmeans


# Show cluster centers for best k
centers_original = scaler.inverse_transform(best_kmeans.cluster_centers_)
print(f"\nBest KMeans: k={best_k}, silhouette={best_silhouette:.3f}, calinski={best_
print("Cluster centers (original scale):")
print(pd.DataFrame(centers_original, columns=df1.columns))

# Assign best clustering to df1
df1["cluster_kmeans"] = best_labels
```

```
KMeans (k=2): silhouette=0.521, calinski=490.81, time=0.0885s
KMeans (k=3): silhouette=0.696, calinski=1066.58, time=0.0273s
KMeans (k=4): silhouette=0.758, calinski=1611.26, time=0.0285s
KMeans (k=5): silhouette=0.805, calinski=3671.36, time=0.0454s
KMeans (k=6): silhouette=0.633, calinski=3198.10, time=0.0592s
KMeans (k=7): silhouette=0.446, calinski=2860.80, time=0.0604s

Best KMeans: k=5, silhouette=0.805, calinski=3671.36
Cluster centers (original scale):
         Age        Income  SpendingScore       Savings
0  59.955414  72448.063694       0.771518   6889.972190
1  87.775510  27866.102041       0.328800  16659.261445
```

```
   2  32.777778  105265.809524       0.309926  14962.778066
   3  24.180000  128029.120000       0.896892   4087.520309
   4  86.000000  119944.040000       0.068378  14808.683793
```

1.2: Clustering Algorithm #2

```
import pandas as pd
from sklearn.cluster import DBSCAN
from sklearn.preprocessing import StandardScaler

# 1. Feature scaling (important for DBSCAN)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(df1)

# 2. Try different DBSCAN parameters and compare metrics
dbscan_results = []
best_silhouette = -1
best_calinski = -1
best_params = None

for eps in [0.15, 0.2, 0.25, 0.3, 0.35]:
    for min_samples in [3, 5, 8, 10]:
        start_dbscan = time.time()
        dbscan = DBSCAN(eps=eps, min_samples=min_samples)
        labels = dbscan.fit_predict(X_scaled)
        end_dbscan = time.time()
        # Only score if more than 1 cluster and not all noise
        n_clusters = len(set(labels)) - (1 if -1 in labels else 0)
        if n_clusters > 1 and np.any(labels != -1):
            sil = silhouette_score(X_scaled[labels != -1], labels[labels != -1])
            cal = calinski_harabasz_score(X_scaled[labels != -1], labels[labels != -
        else:
            sil = float("nan")
            cal = float("nan")
        dbscan_results.append(
            {
                "eps": eps,
                "min_samples": min_samples,
                "n_clusters": n_clusters,
                "silhouette": sil,
                "calinski": cal,
                "time": end_dbscan - start_dbscan,
            }
        )
        print(
            f"DBSCAN (eps={eps}, min_samples={min_samples}): "
            f"clusters={n_clusters}, silhouette={sil:.3f}, calinski={cal:.2f}, "
            f"time={end_dbscan - start_dbscan:.4f}s"
        )
        if sil > best_silhouette:
            best_silhouette = sil
            best_calinski = cal
            best_params = (eps, min_samples)
            best_labels = labels


# Assign best clustering to df1
df1["cluster_dbscan"] = best_labels

# 3. View clusters
cluster_centers_original = (
    df1[df1["cluster_dbscan"] != -1].groupby("cluster_dbscan").mean()  # exclude noi
)

print("\nBest DBSCAN params: eps={}, min_samples={}".format(*best_params))
print(
    "Best silhouette: {:.3f}, Best calinski: {:.2f}".format(
        best_silhouette, best_calinski
    )
)

print("Cluster centers (original scale):")
cols_to_exclude = ["Age", "Income", "SpendingScore", "Savings"]
cols = [col for col in cluster_centers_original.columns if col in cols_to_exclude]
print(cluster_centers_original[cols])
```

```
DBSCAN (eps=0.15, min_samples=3): clusters=19, silhouette=0.069, calinski=994.28
DBSCAN (eps=0.15, min_samples=5): clusters=9, silhouette=0.502, calinski=1748.17
DBSCAN (eps=0.15, min_samples=8): clusters=5, silhouette=0.491, calinski=2261.16
DBSCAN (eps=0.15, min_samples=10): clusters=2, silhouette=0.491, calinski=34.71,
DBSCAN (eps=0.2, min_samples=3): clusters=11, silhouette=0.431, calinski=1950.08
DBSCAN (eps=0.2, min_samples=5): clusters=6, silhouette=0.828, calinski=3403.19,
DBSCAN (eps=0.2, min_samples=8): clusters=3, silhouette=0.878, calinski=6219.44,
DBSCAN (eps=0.2, min_samples=10): clusters=3, silhouette=0.882, calinski=5946.21
DBSCAN (eps=0.25, min_samples=3): clusters=6, silhouette=0.713, calinski=3746.11
DBSCAN (eps=0.25, min_samples=5): clusters=5, silhouette=0.857, calinski=4623.12
DBSCAN (eps=0.25, min_samples=8): clusters=6, silhouette=0.815, calinski=3578.83
DBSCAN (eps=0.25, min_samples=10): clusters=4, silhouette=0.866, calinski=4996.0
DBSCAN (eps=0.3, min_samples=3): clusters=5, silhouette=0.851, calinski=4676.46,
DBSCAN (eps=0.3, min_samples=5): clusters=5, silhouette=0.851, calinski=4694.82,
DBSCAN (eps=0.3, min_samples=8): clusters=5, silhouette=0.852, calinski=4632.53,
DBSCAN (eps=0.3, min_samples=10): clusters=5, silhouette=0.853, calinski=4452.92
DBSCAN (eps=0.35, min_samples=3): clusters=5, silhouette=0.849, calinski=4691.67
DBSCAN (eps=0.35, min_samples=5): clusters=5, silhouette=0.849, calinski=4704.66
DBSCAN (eps=0.35, min_samples=8): clusters=5, silhouette=0.850, calinski=4694.68
DBSCAN (eps=0.35, min_samples=10): clusters=5, silhouette=0.850, calinski=4679.3

Best DBSCAN params: eps=0.2, min_samples=10
Best silhouette: 0.882, Best calinski: 5946.21
Cluster centers (original scale):
```

```
                     Age         Income  SpendingScore       Savings
cluster_dbscan
0              59.340000   71930.640000       0.767922   6896.467039
1              87.831579   27575.915789       0.334299  16706.533097
2              32.160000  105828.080000       0.307794  14697.071919
```

1.3 Model Comparison

While DBSCAN achieved a higher silhouette score (0.882) and Calinski–Harabasz index (5946.21) than the best KMeans model, its output consisted of only three clusters, potentially oversimplifying the customer segmentation and losing nuance in shopper diversity. DBSCAN also requires careful parameter tuning (eps and min_samples) and can be sensitive to scaling, which can make it less straightforward to deploy in a production setting. In contrast, the best KMeans model with k=5 produced well-separated clusters with strong scores (silhouette of 0.805, Calinski–Harabasz of 3671.36), offered more granular customer groupings, and maintained the advantages of speed, simplicity, and interpretability through its clear centroids. Given these factors, KMeans is the better choice in this case because it balances good clustering quality with operational ease and actionable segmentation detail.

1.4 Personas

Below are the personas we believe might correspond to each of our k-means clusters:

Cluster 0 - Middle-aged, stable spenders Average age is about 60, with moderate income of roughly $72,000, relatively high spending scores near 0.77, and savings around $6,900. This group likely represents established professionals or semi-retirees who have steady financial resources and are willing to spend on jewellery without overspending. An example customer might be a 58-year-old office manager with a comfortable salary who purchases jewellery regularly for anniversaries and gifts.

Cluster 1 - Wealth-conserving seniors This is the oldest cluster, with an average age close to 88 and low-to-moderate income around $27,900 but the highest average savings of about $16,700. Their spending score is modest at about 0.33, suggesting they are financially cautious and selective in purchases. A typical member might be an 85-year-old retiree living on investments who only buys high-quality pieces for special family events.

Cluster 2 - Affluent mid-career professionals These customers average 33 years old, have high incomes around $105,000, and solid savings near $15,000, but a lower spending score of about 0.31. They may be career-focused individuals with disposable income who make jewellery purchases infrequently, perhaps for major milestones. An example is a 35-year-old lawyer saving for a house, who buys one or two statement pieces per year.

Cluster 3 - Young, high-spending urbanites The youngest cluster, averaging 24 years old, has the highest income of about $128,000 and the highest spending score near 0.90 but relatively low savings around $4,100. This group may be early-career professionals in lucrative industries who prioritize lifestyle and self-expression over saving. A typical persona could be a 25-year-old tech consultant who enjoys luxury shopping and buys jewellery on impulse.

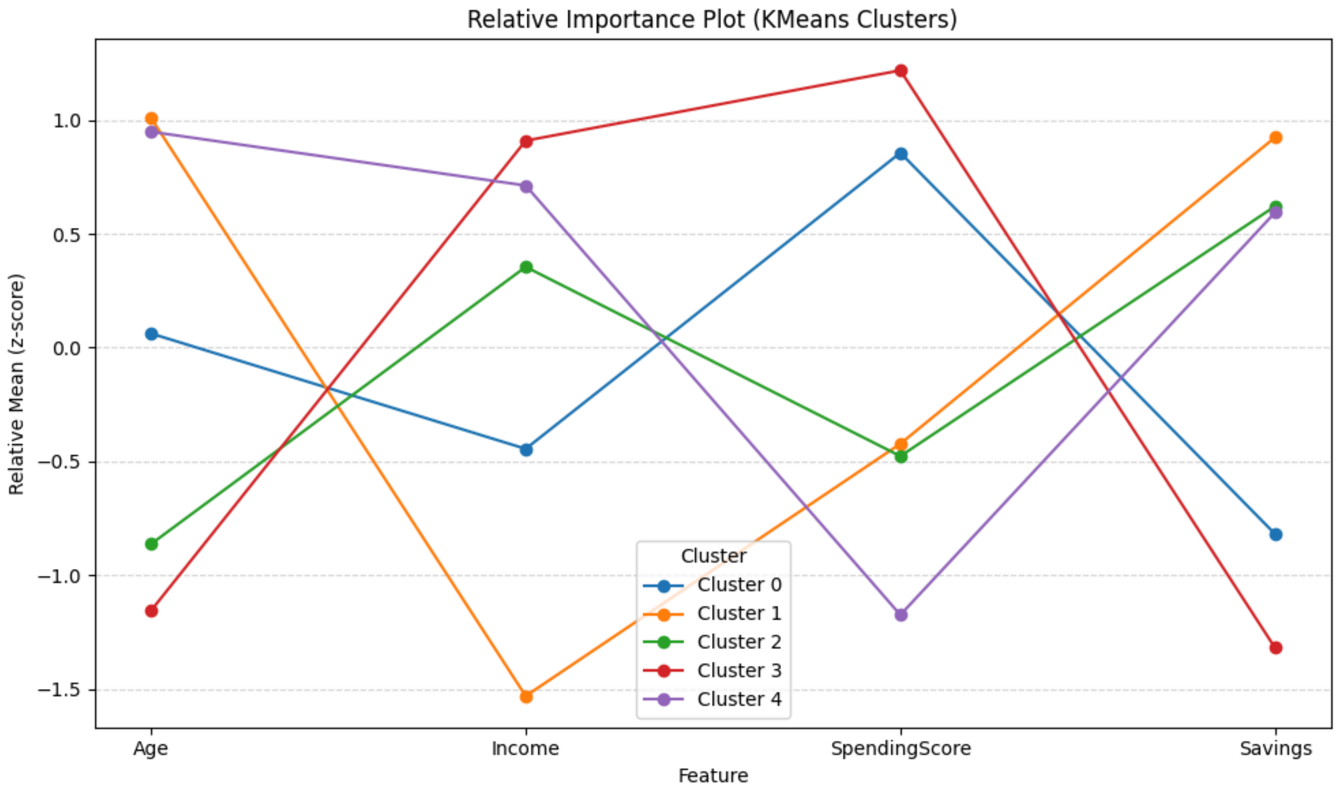
Cluster 4 - Wealthy but frugal elders This group is older with an average age of 86, very high income near $120,000, and substantial savings of about $14,800 but the lowest spending score of roughly 0.07. They are likely financially secure yet disinclined to purchase jewellery frequently, perhaps due to shifting priorities in later life. An example is an 87-year-old retired executive with significant assets who rarely shops for personal items.

```
import matplotlib.pyplot as plt

# Calculate cluster means for each feature (original scale)
cluster_means = pd.DataFrame(centers_original, columns=df1.columns[:-2])

# Normalize means for relative importance (z-score across clusters for each feature)
cluster_means_norm = (cluster_means - cluster_means.mean()) / cluster_means.std()

# Plot
plt.figure(figsize=(10, 6))
for i in range(cluster_means_norm.shape[0]):
    plt.plot(
        cluster_means_norm.columns,
        cluster_means_norm.iloc[i],
        marker="o",
        label=f"Cluster {i}",
    )
plt.title("Relative Importance Plot (KMeans Clusters)")
plt.xlabel("Feature")
plt.ylabel("Relative Mean (z-score)")
plt.legend(title="Cluster")
plt.grid(True, axis="y", linestyle="--", alpha=0.5)
plt.tight_layout()
plt.show()
```



Question 2: Uncle Steve's Fine Foods

2.1: A rule that might have high support and high confidence.

`{eggs} -> {milk}` — Many people buying eggs also grab milk, and both are popular staples. Likely uninteresting, as Steve already knows these basics sell well together.

2.2: A rule that might have reasonably high support but low confidence.

`{bread} -> {jam}` — Bread sells a lot, but only a fraction of buyers add jam. Could be mildly useful for cross-promotions, but not surprising.

2.3: A rule that might have low support and low confidence.

`{flower bouquet} -> {frozen pizza}` — Rarely bought together, and even when bouquets are sold, pizza isn't a common add-on. Not useful for Steve.

2.4: A rule that might have low support and high confidence.

`{birthday cake} -> {birthday card}` — Few transactions involve cakes, but when they do, cards are often purchased too. Useful for bundling and display placement.

Question 3: Uncle Steve's Credit Union

3.0: Load data and split

```
# DO NOT MODIFY THIS CELL

# First, we'll read the provided labeled training data
df3 = pd.read_csv("https://drive.google.com/uc?export=download&id=1wOhyCnvGeY4jplxI8
df3.info()
df3.to_csv('./q3.csv')

from sklearn.model_selection import train_test_split

X = df3.drop('BadCredit', axis=1) #.select_dtypes(['number'])
y = df3['BadCredit']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_stat
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6000 entries, 0 to 5999
Data columns (total 17 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   UserID             6000 non-null   object
 1   Sex                6000 non-null   object
 2   PreviousDefault    6000 non-null   int64
 3   FirstName          6000 non-null   object
 4   LastName           6000 non-null   object
 5   NumberPets         6000 non-null   int64
 6   PreviousAccounts   6000 non-null   int64
 7   ResidenceDuration  6000 non-null   int64
 8   Street             6000 non-null   object
 9   LicensePlate       6000 non-null   object
 10  BadCredit          6000 non-null   int64
 11  Amount             6000 non-null   int64
 12  Married            6000 non-null   int64
 13  Duration           6000 non-null   int64
 14  City               6000 non-null   object
 15  Purpose            6000 non-null   object
 16  DateOfBirth        6000 non-null   object
dtypes: int64(8), object(9)
memory usage: 797.0+ KB
```

3.1: Baseline model

```
# ## 3.1: Baseline model
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.model_selection import cross_val_score

# Identify categorical and numerical columns
categorical_cols = X_train.select_dtypes(include=["object", "category"]).columns.tol
numerical_cols = X_train.select_dtypes(include=["number"]).columns.tolist()

# Preprocessing pipeline
preprocessor = ColumnTransformer(
    transformers=[
        ("num", StandardScaler(), numerical_cols),
        ("cat", OneHotEncoder(handle_unknown="ignore"), categorical_cols),
    ]
)

# Logistic Regression pipeline
clf = Pipeline(
    steps=[
        ("preprocessor", preprocessor),
        ("classifier", LogisticRegression(max_iter=1000, random_state=42)),
    ]
)

# Evaluate with cross-validation (use ROC AUC as metric)
scores = cross_val_score(clf, X_train, y_train, cv=5, scoring="roc_auc")
print(f"Baseline Logistic Regression ROC AUC (mean over 5 folds): {scores.mean():.3f
```

```
Baseline Logistic Regression ROC AUC (mean over 5 folds): 0.835
```

3.2: Adding feature engineering

```
# Create 'Age' feature

# Assume application_date is the same for all: use latest date in dataset or a fixed
application_date = pd.to_datetime("2025-07-01")

X_train = X_train.copy()
X_test = X_test.copy()

# Convert DateOfBirth to datetime if not already
X_train["DateOfBirth"] = pd.to_datetime(X_train["DateOfBirth"])
X_test["DateOfBirth"] = pd.to_datetime(X_test["DateOfBirth"])

# Calculate age in years (approximate)
X_train["Age"] = (application_date - X_train["DateOfBirth"]).dt.days // 365
X_test["Age"] = (application_date - X_test["DateOfBirth"]).dt.days // 365

# Create 'FinancialCommitmentsRatio' feature. This is a simple ratio of financial co
X_train["FinancialCommitmentsRatio"] = (
    X_train["PreviousAccounts"] + X_train["NumberPets"]
) / X_train["ResidenceDuration"].replace(0, np.nan)
X_test["FinancialCommitmentsRatio"] = (
    X_test["PreviousAccounts"] + X_test["NumberPets"]
) / X_test["ResidenceDuration"].replace(0, np.nan)

# Create 'MaritalStability' feature. This is a simple interaction term between marit
X_train["MaritalStability"] = (X_train["Married"] == "yes").astype(int) * X_train["R
X_test["MaritalStability"] = (X_test["Married"] == "yes").astype(int) * X_test["Resi
```

3.3: Adding feature selection

```
from sklearn.feature_selection import SelectFromModel
from sklearn.ensemble import RandomForestClassifier
from sklearn.impute import SimpleImputer



# Re-identify columns after feature engineering
categorical_cols = X_train.select_dtypes(include=["object", "category"]).columns.tol
numerical_cols = X_train.select_dtypes(include=["number"]).columns.tolist()

# Preprocessing pipeline (same as before)
preprocessor = ColumnTransformer(
    transformers=[
        ("num", StandardScaler(), numerical_cols),
        ("cat", OneHotEncoder(handle_unknown="ignore"), categorical_cols),
    ]
```

```
)

# Feature selection using RandomForestClassifier as estimator
feature_selector = SelectFromModel(RandomForestClassifier(n_estimators=100, random_s

# Pipeline with preprocessing, feature selection, and classifier
clf_fs = Pipeline(
    steps=[
        ("preprocessor", preprocessor),
        ("impute", SimpleImputer(strategy="mean")),
        ("feature_selection", feature_selector),
        ("classifier", LogisticRegression(max_iter=1000, random_state=42)),
    ]
)

# Evaluate with cross-validation (ROC AUC)
scores_fs = cross_val_score(clf_fs, X_train, y_train, cv=5, scoring="roc_auc")
print(f"Logistic Regression with Feature Selection ROC AUC (mean over 5 folds): {sco
```

```
Logistic Regression with Feature Selection ROC AUC (mean over 5 folds): 0.919
```

3.4: Adding hyperparameter tuning

```
from sklearn.model_selection import GridSearchCV

# Define parameter grid for LogisticRegression and RandomForest feature selector
param_grid = {
    "feature_selection__estimator__n_estimators": [50, 100, 200],
    "classifier__C": [0.001, 0.01, 0.1, 1, 10],
    "classifier__penalty": ["l2", "l1"],
    "classifier__solver": ["liblinear"],
}

# Reuse pipeline from previous step (clf_fs)
grid_search = GridSearchCV(
    clf_fs,
    param_grid,
    cv=5,
    scoring="roc_auc",
    n_jobs=-1,
    verbose=1,
)

grid_search.fit(X_train, y_train)
print(f"Best ROC AUC: {grid_search.best_score_:.3f}")
print("Best parameters:", grid_search.best_params_)
```

```
Fitting 5 folds for each of 30 candidates, totalling 150 fits
Best ROC AUC: 0.925
Best parameters: {'classifier__C': 0.1, 'classifier__penalty': 'l1', 'classifier
```

3.5: Performance estimation on testing data

```
from sklearn.metrics import roc_auc_score, classification_report

# Use the best estimator from grid search
best_model = grid_search.best_estimator_

# Predict probabilities and classes on the test set
y_pred_proba = best_model.predict_proba(X_test)[:, 1]
y_pred = best_model.predict(X_test)

# Evaluate ROC AUC and print classification report
roc_auc = roc_auc_score(y_test, y_pred_proba)
print(f"Test ROC AUC: {roc_auc:.3f}")
print("Classification report on test data:")
print(classification_report(y_test, y_pred))
```

```
Test ROC AUC: 0.935
Classification report on test data:
              precision    recall  f1-score   support

           0       0.91      0.97      0.94       991
           1       0.78      0.57      0.66       209

    accuracy                           0.90      1200
   macro avg       0.85      0.77      0.80      1200
weighted avg       0.89      0.90      0.89      1200
```

Our optimized logistic regression model achieved strong performance, with a test ROC AUC of 0.935—an improvement from the baseline score of 0.835 prior to feature engineering, feature selection, and hyperparameter tuning. On the test set, it reached an overall accuracy of 89%, with high precision and recall for the majority class (class 0) and more modest recall for the minority class (class 1). The best configuration used L1 regularization with C = 0.1, the liblinear solver, and feature selection via a tree-based estimator with 200 trees.

Question 4: Uncle Steve's Wind Farm

Using no model, Uncle Steve currently pays for repairs only when a turbine actually fails. Based on the confusion matrices, there are 256 actual failures in the data, which at $20,000 per repair costs

$5,120,000 per year. When a predictive model is used, any turbine predicted to fail is inspected for $500.

If it is truly about to fail, it is then serviced immediately for an additional $2,000. If the model misses a failure, the turbine breaks and the full $20,000 repair cost applies. For the random forest, the cost is 201 x $2,500 for true positives, plus 50 x $500 for false positives, plus 55 x $20,000 for false negatives, which totals $1,627,500. For the RNN, the cost is 226 x $2,500 for true positives, plus 1,200 x $500 for false positives, plus 30 x $20,000 for false negatives, which totals $1,765,000. Compared to the current cost of $5,120,000, the random forest saves $3,492,500 while the RNN saves $3,355,000. Although the RNN identifies more actual failures, it produces far more false positives, which leads to many unnecessary inspections. This extra cost outweighs the benefit of its higher recall. In this case, the random forest is the better choice because its higher precision results in greater overall savings.