(Final Draft) Shakar_Credit_Approval_FinalProject

April 23, 2025

Introduction: (Ignore hidden/commented markdown cells)
(Also note that some printouts in the code may contradict what I wrote in the PDF because I've ran them again in the code for submission since I commented them in the PDF for space)

We're using the UCI Credit Approval Dataset https://archive.ics.uci.edu/dataset/27/credit+approval and we'll build a Logistic Regression Model that generalizes well and can be used to classify whether if a person applying for credit will be aproved or not in a relatively accurate manner. We're using Logistic Regression because it does a good enough job for models that have binary target variables that's 0 or 1 (rejected or approved for credit)

I'm interested in this topic because I'm interested in finance and credit is very important in facilitating business transactions, consumer spending, investments, and so on.

Preparing the dataset before training:

[2]:
```
# Assign columns A1 to A16
df = pd.read_csv('./crx.data', header=None, na_values="?")
df.columns = [f"A{i+1}" for i in range(df.shape[1])]
categorical_preview_cols = df.select_dtypes(include=['object']).columns.tolist()

# Strip and map only valid class values and map +/- into 1/0
df["A16"] = df["A16"].astype(str).str.strip().map({'+': 1, '-': 0})
print("\n",df.shape[0], "rows ×", df.shape[1], "columns")
```

```
690 rows × 16 columns
```

[3]:
```
# Check how many Missing (NaN) values rows there are for each column/predictor␣
 ↪variables
df_nas = df.isna().sum()
# print(df_nas[df_nas>0]) # commented to save space
# we only lose at least 13 out of 690 rows that have missing values, so we drop␣
 ↪them
df_clean = df.dropna()
print("\n",df_clean.shape[0], "rows ×", df_clean.shape[1], "columns")
# df_clean.head() # # commented to save space
```

```
653 rows × 16 columns
```

By checking class distributions (commented to save space, check code), we see that 26 0's and 11 1's are dropped (37 rows / 690 rows = approx 5.36%)
Dropping NA's cause very minimal shift toward approvals. Likely acceptable to drop NaN rows for most models. Since the class shift is negligible after dropping NaN rows, dropping NAs is simpler for our model building hereon out.

[4]:
```
# If keeping the dropped version (653 rows)
df_clean = df.dropna()
# Numeric columns & Categorical columns
num_cols = ['A2', 'A3', 'A8', 'A11', 'A14', 'A15']; cat_cols = ['A1', 'A4',␣
 ↪'A5', 'A6', 'A7', 'A9', 'A10', 'A12', 'A13']
```

[5]:
```
df_clean = df.dropna()
print(df_clean['A16'].value_counts(normalize=True))
```

```
A16
0    0.546708
1    0.453292
Name: proportion, dtype: float64
```

[6]:
```
# Drop rows with NAs
df_clean = df.dropna().copy()  # Now has 653 rows (from 690)
# Separate features (X) and target (y)
X = df_clean.drop('A16', axis=1)  # All predictor columns
y = df_clean['A16']               # Target (0=denied, 1=approved)
```

[7]:
```
# X.dtypes # uncomment if needed
```

To preprocess our data we use One-Hot Encoding to include categorical variables. Categorical Columns (based on UCI dataset description):
A1, A4, A5, A6, A7, A9, A10, A12, A13. (Note: Verify by checking dtypes with X.dtypes for "object" dtypes). One-hot encoding creates binary (0/1) features for each category and we use drop='first' to drop the first binary column in order to avoid multicollinearity.

[8]:
```
# Categorical columns (adjust based on your dataset) & Numeric columns␣
 ↪(standardize later)
categorical_cols = ['A1', 'A4', 'A5', 'A6', 'A7', 'A9', 'A10', 'A12', 'A13'];␣
 ↪numeric_cols = X.columns.difference(categorical_cols).tolist()

# ColumnTransformer for encoding categoricals and keep numericals unchanged/
 ↪untransformed.
preprocessor = ColumnTransformer(transformers=[('cat',␣
 ↪OneHotEncoder(drop='first'), categorical_cols),
        ('num', 'passthrough', numeric_cols)])
X_processed = preprocessor.fit_transform(X)
```

We standardize numeric features to ensure equal feature scaling of the numeric features, since some features may be large such as for example income, etc. that often have different units in the thousands, even hundreds of thousands. Standardizing puts them on the same scale (mean=0,

std=1), in order to prevent features with larger magnitudes from dominating the model.
Also, models like neural networks use gradient descent, which converges faster when features are standardized.

[9]:
```
# Standardize numeric features (after one-hot encoding)
scaler = StandardScaler()
X_processed[:, len(preprocessor.named_transformers_['cat'].
 ↪get_feature_names_out()):] = \
   scaler.fit_transform(X_processed[:, len(preprocessor.
 ↪named_transformers_['cat'].get_feature_names_out()):])
```

Train-Test Split:

[10]:
```
X_train, X_test, y_train, y_test = train_test_split(X_processed, y, test_size=0.
 ↪3, random_state=69, stratify=y)
# stratify = y is to evenly distribute the values of y in each split.
```

Build the initial Neural Network on the whole dataset using all variables & Train the initial Model:
1st hidden layer: 64 neurons ReLU activation function to learn complex patterns from input features.
2nd hidden layer: 32 neurons ReLU to refine the learned representations.
Both with 30% Dropout, which randomly sets 30% of the layer's outputs to 0 during training. We use a moderate 30% to prevent overfitting to training data noise, any higher would risk underfitting and any lower may not prevent overfitting.
Sigmoid activation in the outer layer, which makes the outputs to [0,1] range, which is what we need for binary classification (credit approval: 0=reject, 1=approve). OPTONAL: We refrain from using normalization since Loss and accuracy seems relatively stable without it.
Our model has: 1st Hidden layer for processing the input data (aka. Input layer): 64 neurons [expecting 37 features, so 2432 params = (37 inputs * 64 neurons) + 64 biases]
2nd Hidden layer: 32 neurons, so 2080 params = (64 inputs * 32 neurons) + 32 biases
Output Layer: 1 neuron, so 33 params = 32 inputs * 1 neuron + 1 Bias (sigmoid for binary classification)
Total params: 4,545

We use learning rate = 0.001 since it is the standard initial learning rate for the Adam optimizer because it's small enough to avoid overshooting optimal weights (divergence) and large enough to make meaningful progress without being impractically slow. Can Decrease if loss oscillates or diverges.
And we use binary_crossentropy to measure the difference between predicted probabilities (sigmoid outputs) and true binary labels (0 or 1).

For training:
To prevent overfitting we use EarlyStopping to stop training when the model stops improving on validation data.
Key Parameters:
patience=5: the model will wait and run for 5 epochs without improvement before stopping the training. It monitors val_loss by default (or another metric if specified). If there's no improvement for 5 patience epochs, training will stop.

restore_best_weights=True: Revert to the model weights from the epoch with the best validation performance when we do stop training.

We choose a reasonable 50 epochs for a small to medium sized dataset, this is sufficient for convergence without overfitting.
And 32 batch size as smaller batches offer more frequent updates, helping your model escape poor local minima, 32 is best balance of stability and efficiency as it has smoother gradients than lower batch sizes, i.e. faster convergence as it handles noise better, although it comes at a slightly higher risk of overfitting. 16 would be better for smaller datasets and more complex models, whereas 32 is fine for medium sized and simpler models.
And we address the class imbalance during training by making the model pay more attention to the under-represented class using `class_weight={0: 0.55, 1: 0.45}`, since otherwise the model may be biased towards one class. Weights adjust the loss function to penalize errors on the minority class more heavily.

[11]:
```
# Get input shape (number of features after encoding)
input_shape = X_train.shape[1]
model = Sequential([
    Dense(64, activation='relu', input_shape=(input_shape,)),Dropout(0.3),  #␣
 ↪Reduce overfitting
    Dense(32, activation='relu'), Dropout(0.3), Dense(1, activation='sigmoid') ␣
 ↪# Binary classification
])
# Compile the model
model.compile(optimizer=Adam(learning_rate=0.
 ↪001),loss='binary_crossentropy',metrics=['accuracy', tf.keras.metrics.
 ↪AUC(name='auc')]
)
callbacks = [EarlyStopping(patience=5, restore_best_weights=True)]
history = model.fit(X_train, y_train, validation_data=(X_test, y_test),␣
 ↪epochs=50, batch_size=32, class_weight={0: 0.55, 1: 0.45},
                    callbacks = callbacks, verbose = 0 ) # class_weight to␣
 ↪adjust for class imbalance
```

```
WARNING:absl:At this time, the v2.11+ optimizer `tf.keras.optimizers.Adam` runs
slowly on M1/M2 Macs, please use the legacy Keras optimizer instead, located at
`tf.keras.optimizers.legacy.Adam`.
```

Evaluate the Model:

[12]:
```
# Evaluate on test data
loss, accuracy, auc = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {accuracy:.2f}")
print(f"AUC: {auc:.2f}")  # AUC > 0.8 is good, ideally we want > 0.9 which is␣
 ↪excellent
# Predictions
y_pred = (model.predict(X_test) > 0.5).astype(int) # threshold > 0.5 default␣
 ↪for binary classifications
```

```
7/7 [==============================] - 0s 872us/step - loss: 0.2986 - accuracy:
0.8622 - auc: 0.9454
Test Accuracy: 0.86
AUC: 0.95
7/7 [==============================] - 0s 568us/step
```

Comment: AUC = 0.9454 > 0.9: Excellent at discriminating between the two classes (far better than random guessing (0.5)). Test Accuracy approx. 86%: Good, Model generalizes well to unseen data. May be possible to optimize further.

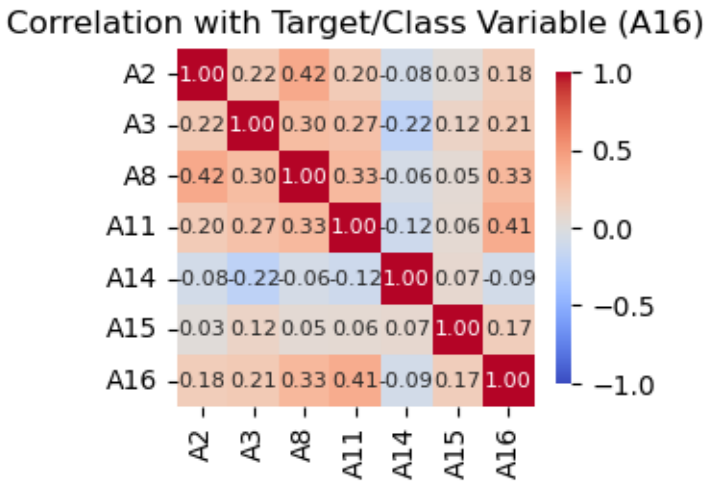
[13]:
```
# Get weights from the first dense layer
weights = model.layers[0].get_weights()[0]  # Shape: (n_features, 64)
avg_impact = np.mean(np.abs(weights), axis=1)
# Get feature names (after one-hot encoding)
feature_names = np.concatenate([preprocessor.named_transformers_['cat'].
 ↪get_feature_names_out(), numeric_cols])
```

So far we trained on ALL of the variables, now let's check the Variable Importance OR Signifcance to see which variables we can drop from training to see if we can simplify the model more.
We can check this by finding which features have larger absolute weights, therefore contribute more vs which have lower absolute weights, i.e. contribute less.

Top & Bottom Features (High & Low Avg_Weight) (some of these are likely to change when re-running the model): the top features above strongly influence the model's predictions. For example: A9_t (Likely Employment Status: "t" category) has the highest impact.
Bottom Features (Low Avg_Weight) contribute minimally to predictions (but may still have subtle effects).
Numeric Features: such as num___A15 are moderately important.

We Also want to check Correlation Analysis (Numerical Features) First, let's check correlations between numeric variables and the target (A16):

[14]:
```
# print("Columns in df_clean:", df_clean.columns.tolist())  # Should include␣
 ↪'A16'
# Select numeric features (adjust based on your dataset)
numeric_cols = ['A2', 'A3', 'A8', 'A11', 'A14', 'A15']  # Numeric columns
df_numeric = df_clean[numeric_cols + ['A16']]  # Use 'A16' as target name
# Calculate correlations
corr_matrix = df_numeric.corr()
# Plot heatmap
plt.figure(figsize=(3, 3))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', vmin=-1, vmax=1,fmt=".
 ↪2f",annot_kws={"size": 8},  # Smaller annotation text,
            cbar_kws={"shrink": 0.7},  # Smaller color bar,
            square=True  # Makes cells square (more compact)
           )
plt.title("Correlation with Target/Class Variable (A16)")
plt.show()
```



Interpretation of our feature Correlation with the target variable (A16) Results:
Higher values = more approval. A11 has strongest positive relationship, we Keep this. A8 has Moderate positive impact, 2nd strongest, we also Keep this. A3 and A2 has Weak positive influence. need to investigate further (may or may not keep).
A15 also has weak positive influence. Low priority (we'll consider dropping this as well). A14 has Negligible negative impact. We'll most likely drop this.
Recommended features to keep: A11 (Strongest predictor), A8 (High impact), A3 (Borderline; may or may not keep depending on model performance)
Drop: A14 (Near-zero correlation), A2 and A15 (Weak correlations, unless prior knowledge about the feature/variable suggests otherwise)
Retraining the simpler Model with fewer selected features will give us faster training + easier interpretation, and better gneralization: i.e. removing noise (especially A14) improves robustness.

[16]:
```
selected_features = ['A11', 'A8', 'A3']  # Add categorical features later
X_selected = df_clean[selected_features]; categorical_cols = ['A1', 'A4', 'A5',␣
 ↪'A6', 'A7', 'A9', 'A10', 'A12', 'A13']
```

Checking Correlations for Categorical Variables
Since categorical features (like A1, A4, A7, etc.) are non-numeric, we use target encoding (mean approval rate per category) or statistical tests (Chi-square, Cramer's V) to measure their relationship with the target (A16). Here's how to do it:
Target Encoding (Approval Rate per Category) For each categorical feature, calculate the mean approval rate (A16) for each category. This shows which categories strongly predict approval/denial.

[17]:
```
# # List of categorical columns (adjust based on your dataset)
# THESE values maybe different from expalanations of Markdown cells below,
# this is because this was ran after the PDF
categorical_cols = ['A1', 'A4', 'A5', 'A6', 'A7', 'A9', 'A10', 'A12', 'A13']
for col in categorical_cols: # Commented to save space
    approval_rate = df_clean.groupby(col)['A16'].mean().
 ↪sort_values(ascending=False)
```

```
#     print(f"\n{col}: " + ", ".join([f"{k}={v:.1%}" for k, v in approval_rate.
 ↪items()]))
# print('\n',df.dropna()['A16'].value_counts(normalize = True)) # Commented to␣
 ↪save space
```

Interpretation: Categories with approval rates far from the cleaned dataset mean (~45.3%) are influential.

Example: A9_t has 79.7% approval, A10_t has 70.7% approval (verify on code printouts), etc. therefore A9 and A10 is a critical feature.

Key Insights from Target Encoding Focus on features where:
- Approval rates differ by a conservative amount of **>15%** from the cleaned dataset mean (45.3%), and High variance between categories (compare with print outs in code).

Feature	Key Categories	Severity of Approval Rate	Variance from Mean	Decision
A4	`l` (100%) vs `y` (29.6%)	Extreme	+54.7% / -15.7%	Keep
A5	`gg` (100%) vs `p` (29.6%)	Extreme	+54.7% / -15.7%	Keep
A6	`x` (83.3%) vs `ff` (14.0%)	High	+38.0% / -31.3%	Keep
A7	`z` (75.0%) vs `ff` (14.8%)	High	+29.7% / -30.5%	Keep
A9	`t` (79.7%) vs `f` (5.9%)	High	+34.4% / -39.4%	Keep
A10	`t` (70.7%) vs `f` (25.4)	High	+25.4% / -19.9%	Keep
A1	`a` (46.8%) vs `b` (44.7%)	Minimal	+1.5% / -0.6%	Drop
A12	`t` (48.0) vs `f` (43.0%)	Minimal	+2.7% / -2.3%	Drop
A13	`p` (50%) vs `s` (28.3%)	Moderate	+4.7% / -17.0%	Borderline

Next, Chi-Square Test (Statistical Significance) Tests whether a categorical feature and the target are independent.

[18]:
```
for col in categorical_cols:
    contingency_table = pd.crosstab(df_clean[col], df_clean['A16'])
    chi2, p, _, _ = chi2_contingency(contingency_table)
    print(f"{col}: p-value = {p:.4f}")
```

```
A1: p-value = 0.6734
A4: p-value = 0.0000
A5: p-value = 0.0000
A6: p-value = 0.0000
A7: p-value = 0.0000
A9: p-value = 0.0000
A10: p-value = 0.0000
A12: p-value = 0.2305
A13: p-value = 0.0341
```

Cramer's V (Strength of Association) Measures correlation strength between 0 (no association) and 1 (perfect association).
Key points: p-value < 0.05: The feature and target are correlated (significant). p-value > 0.05: No evidence of correlation.

Chi-Square Test Results
- **Highly Significant ($p < 0.001$):**

A4, A5, A6, A7, A9, A10 → Strong statistical relationship with approval. Keep these. - **Not Significant (p > 0.05):**
A1, A12 → No evidence of relationship. Drop these. - **Marginally Significant:**
A13 (p = 0.0341) → Somewhat significant to Weak relationship. May or may not keep.

[19]:
```
def cramers_v(contingency_table):
    chi2, _, _, _ = chi2_contingency(contingency_table)
    n = contingency_table.sum().sum()
    phi2 = chi2 / n
    r, k = contingency_table.shape
    return np.sqrt(phi2 / min(k-1, r-1))

for col in categorical_cols:
    contingency_table = pd.crosstab(df_clean[col], df_clean['A16'])
    print(f"{col}: Cramer's V = {cramers_v(contingency_table):.3f}")
```

```
A1: Cramer's V = 0.016
A4: Cramer's V = 0.183
A5: Cramer's V = 0.183
A6: Cramer's V = 0.371
A7: Cramer's V = 0.257
A9: Cramer's V = 0.736
A10: Cramer's V = 0.449
A12: Cramer's V = 0.047
A13: Cramer's V = 0.102
```

If we have: 0.1–0.3: Weak association, 0.3–0.5: Moderate association, >0.5: Strong association
Cramer's V Strength
- **Very Strong (V > 0.5):**
A9 (0.736) → Dominant predictor. Keep this.
- **Moderate (0.3 V 0.5):**
A6 (0.371), A10 (0.449) → strong predictors. Keep these. - **Weak (V < 0.3):**
A1, A4, A5, A7, A12, A13 → Less impactful, but some may be important to keep. Weakest ones is A1, A12 and A13, we choose to drop these 3 and keep the rest.

Final Categorical Feature Selection (*Results may vary slightly between runs but should remain relatively consistent, approval rates are found from commented printouts*) **(Keep those with High Impact)**
- **A9 (t/f):** Extreme approval rate difference (80% vs 6%).
- **A6 (x/ff):** 84% vs 13% approval.
- **A10 (t/f):** 71% vs 25% approval.
- **A4 (l/y):** 100% vs 28% approval (*investigate l*).
- **A5 (gg/p):** 100% vs 28% approval (*investigate gg*).
- **A7 (z/ff):** 75% vs 14% approval.

Drop (Low Impact)
- A1, A12: Minimal variance, high p-values, low Cramer's V.
- A13: Borderline, not very impactful. Drop this unless prior knowledge suggests importance. (For example if you know this represents income or credit)

Next steps: Investigate Suspicious Categories: Check why A4_l and A5_gg have 100% approval (possible data issues?).

[20]:
```
print(df_clean[df_clean['A4'] == 'l'][['A4', 'A16']].value_counts())
print(df_clean[df_clean['A5'] == 'gg'][['A5', 'A16']].value_counts())
```

```
A4  A16
l   1      2
Name: count, dtype: int64
A5  A16
gg  1      2
Name: count, dtype: int64
```

[21]:
```
df_clean['A4'] = df_clean['A4'].replace({'l': 'u'})  # Merge 'l' into 'u'
df_clean.loc[:, 'A5'] = df_clean['A5'].replace({'gg': 'g'}) # Merge 'gg' into␣
 ↪'g'
```

Interpretation:
There are only 2 cases where A4 = 'l', and both were approved (A16 = 1). And gg category has only 2 samples as well, (both had 100% approval in our target encoding).
This is statistically unreliable (may be data entry errors or edge cases). This is likely a rare category or data anomaly.
As a result, we merged the A4_l category with another category (e.g., u) and A5_gg into A5_g category as they're too rare to generalize. Model is more stable this way. If we kept them, the model may overfit to these 2 samples, harming generalization.

[22]:
```
selected_categorical = ['A4', 'A5', 'A6', 'A7', 'A9', 'A10']; selected_numeric␣
 ↪= ['A8', 'A11']
# From the correlation analysis we dropped A1, A12 and A13 (categoriclas), and␣
 ↪recall earlier we also dropped A2, A15, A14 (numericals) due to weak␣
 ↪correlations
# We also Dropped A3 because: Weak correlation (0.207  low impact), since our␣
 ↪model already has stronger predictors (A9, A6, A8, etc.).
# Simplifying the model improves interpretability without sacrificing␣
 ↪performance.
```

[23]:
```
selected_features = [
    # Categorical (merged/cleaned)
    'A4', 'A5', 'A6', 'A7', 'A9', 'A10',
    # Numeric (from correlation analysis)
    'A8', 'A11']; X_selected = df_clean[selected_features]; y = df_clean['A16']
```

Preprocessing all features (both categorical and numeric):
Categorical: One-hot encoded (e.g., A4, A5, etc.). Numeric: Standardized (e.g., A8, A11 scaled to mean=0, variance=1).

Chaining preprocessing + training into a single workflow:
Ensure preprocessing steps (like scaling) are reapplied consistently during training and prediction (e.g., on new data). Avoid data leakage (e.g., scaling is fitted only on training data, not the entire dataset).

[24]:
```
# Categorical features (one-hot encode)
categorical_cols = ['A4', 'A5', 'A6', 'A7', 'A9', 'A10']; numeric_cols = ['A8',␣
 ↪'A11']
# # Preprocessor for categorical (one-hot) and numeric (scaling) features
preprocessor = ColumnTransformer(transformers=[('cat',␣
 ↪OneHotEncoder(drop='first'),categorical_cols), ('num', StandardScaler(),␣
 ↪numeric_cols)])
```

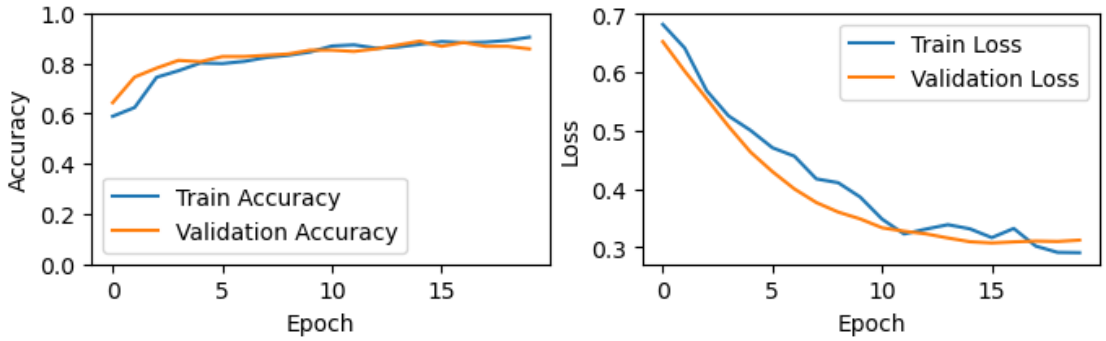
[25]:
```
# Convert to DataFrame with named columns
X_train_df = pd.DataFrame(X_train, columns=feature_names); X_test_df = pd.
 ↪DataFrame(X_test, columns=feature_names)
# Only scale numeric columns (no one-hot needed since they're already␣
 ↪pre-encoded previously)
preprocessor = ColumnTransformer([('num', StandardScaler(), numeric_cols)],␣
 ↪remainder='passthrough') # Keeps already-encoded categoricals as is
X_train_processed = preprocessor.fit_transform(X_train_df); X_test_processed =␣
 ↪preprocessor.transform(X_test_df)
```

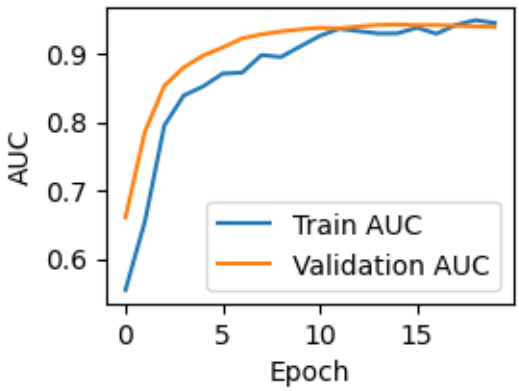
[26]:
```
# Define model (same as last time except with the newly processed data)
model = Sequential([
    Dense(64, activation='relu', input_shape=(X_train_processed.shape[1],)),␣
 ↪Dropout(0.3),
    Dense(32, activation='relu'), Dropout(0.3), Dense(1, activation='sigmoid')
])
# Compile with AUC metric
model.compile(optimizer='adam', loss='binary_crossentropy',␣
 ↪metrics=['accuracy', tf.keras.metrics.AUC(name='auc')]
)
# Early stopping to prevent overfitting.
early_stop = EarlyStopping(monitor='val_auc', patience=5, mode='max',␣
 ↪restore_best_weights=True)
```

[27]:
```
# Convert sparse output to dense arrays to Preprocess data (ensure dense output)
history = model.fit(X_train_processed, y_train,␣
 ↪validation_data=(X_test_processed, y_test), epochs=50, batch_size=32,␣
 ↪callbacks=[early_stop], verbose=0 )
# Evaluate on test data
loss, accuracy, auc = model.evaluate(X_test_processed, y_test, verbose=0)
print(f"Test Loss: {loss:.3f}")
print(f"Test Accuracy: {accuracy:.3f}")
print(f"Test AUC: {auc:.3f}")
```

```
Test Loss: 0.309
Test Accuracy: 0.888
Test AUC: 0.943
```

[28]:
```
# Plot training vs test/validation history
plt.figure(figsize=(8, 2)); plt.subplot(1, 2, 1) # Accuracy plot
plt.plot(history.history['accuracy'], label='Train Accuracy'); plt.plot(history.
 ↪history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch'); plt.ylabel('Accuracy'); plt.ylim([0, 1])  # Set y-axis to␣
 ↪0-1 for accuracy
plt.legend()

plt.subplot(1, 2, 2) # Loss plot
plt.plot(history.history['loss'], label='Train Loss'); plt.plot(history.
 ↪history['val_loss'], label='Validation Loss')
plt.xlabel('Epoch'); plt.ylabel('Loss'); plt.legend()

plt.figure(figsize=(6,2)); plt.subplot(1, 2, 1);  plt.plot(history.
 ↪history['auc'], label='Train AUC')
plt.plot(history.history['val_auc'], label='Validation AUC'); plt.
 ↪xlabel('Epoch'); plt.ylabel('AUC')
plt.legend()
plt.show()
```





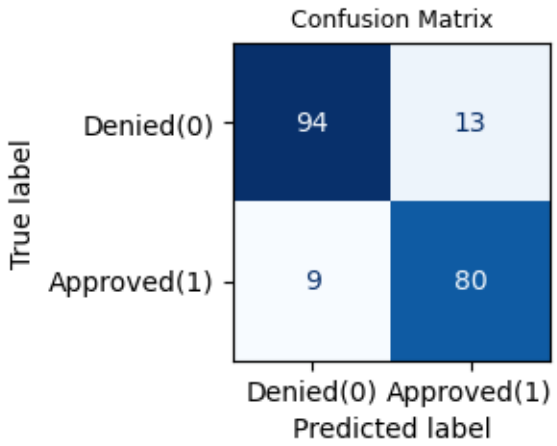
Performance Summary
Last time with the full data model we had: loss: 0.2986 - accuracy: 0.8622 - auc: 0.9454 Test Accuracy: 0.86 AUC: 0.95
This time we have Test Loss 0.309 (goes down smoother but slightly higher loss than last time which is okay [compare to plot of training on full data on code]), Test Accuracy 0.888 (strong for credit approval tasks) and slightly higher which is good, and AUC 0.943 $<$ 0.9454 last time (Excellent discrimination between approvals/denials [AUC $>$ 0.9 is ideal]). So we have a slight downgrade with this simpler model since better separation, but it's still generalizes well with this much simpler model, which is great..
Therefore, this simpler model generalizes well enough. Also notice that this model stops in 15-20 epochs, so further training likely won't improve the model.

Comments: Initial sharp increase in improvement of accuracy where there's rapid learning of easy patterns (such as very obvious and clear-cut approval/rejection cases). Then after a few epochs we have gradual increase : where the model learns finer, more complex patterns. The convergence of the gap between training and validation(test) accuracy after a certain number of epochs implies reduced overfitting as the model stabilizes, the initial divergence in gap of accuracy is normal as the model transitions from memorizing noise to learning generalizable patterns.

Test Loss seems stable enough and steady, whereas accuracy is slightly rough (not very smooth) due to slight fluctuations due to randomness but it's still relatively stable enough to learn patterns to generalize well, which is good.
There's Minimal Overfitting: The gap between training and validation accuracy is small, indicating your model generalizes well.
Strong Performance: For credit approval (a noisy real-world problem), ~88.8% accuracy and AUC 0.943 are highly competitive.

[29]:
```
y_pred = (model.predict(X_test_processed) > 0.5).astype(int)
cm = confusion_matrix(y_test, y_pred)
# Create a tiny figure FIRST
fig, ax = plt.subplots(figsize=(3.1, 3.1))  # Adjusted to 3x3 for readability
# Plot directly onto the axis
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=['Denied(0)',␣
 ↪'Approved(1)'])
disp.plot(ax=ax, cmap='Blues', colorbar=False)  # Disable colorbar to save space
plt.title('Confusion Matrix', fontsize=9)  # Smaller title
plt.tight_layout()  # Prevent label clipping
plt.show()
```

```
7/7 [==============================] - 0s 745us/step
```



[30]:
```
TN, FP, FN, TP = cm.ravel()
print(TN, FP, FN, TP, '\n')
# Calculate critical metrics
accuracy = (TP + TN) / (TP + TN + FP + FN); precision = TP / (TP + FP)  # % of␣
 ↪approvals that were correct
sens = TP / (TP + FN)     # % of true approvals correctly identified
f1 = 2 * (precision * sens) / (precision + sens)
print(f"Accuracy: {accuracy:.3f}")
print(f"Precision: {precision:.3f} (Avoids bad loans)")
print(f"Sensitivity: {sens:.3f} ( High sensitivity, fewer good clients␣
 ↪rejected, avoids missing good clients)")
print(f"Balanced f1-score: {f1:.3f} (Balances precision/sensitivity)")
```

```
94 13 9 80

Accuracy: 0.888
Precision: 0.860 (Avoids bad loans)
Sensitivity: 0.899 ( High sensitivity, fewer good clients rejected, avoids
missing good clients)
Balanced f1-score: 0.879 (Balances precision/sensitivity)
```

Key Metrics Breakdown:
Accuracy: (TP + TN) / Total = 88.8% of all predictions are correct.
Precision: TP / (TP + FP) = 86.0% of approved loans are good (14.0% may default). Sensitivity: TP / (TP + FN) = 89.9% of truly creditworthy applicants are approved (10.1% are wrongly denied). Balanced F1-Score $2(Prec \text{Sens})/(\text{Prec}+\text{Sens}) = 0.879$ Balanced measure of precision and sensitivity.

Business Implications:
False Positives (FP) tells us there are 13 Bad loans (approved risky applicants) so we need to increase decision threshold (such as approve only if probability > 0.7 instead of 0.5). False Negatives (FN) tells us there are 9 Good loans that are Lost revenue (rejected good clients), so we need to lower

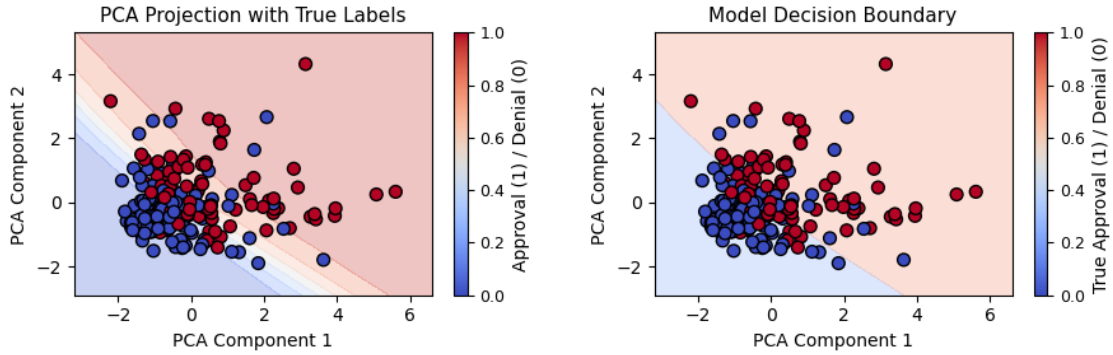
threshold or improve model for edge cases.

PCA and Decision Boundary: Since our credit approval model uses multiple features (both numeric and categorical), we can't directly visualize a decision boundary in raw feature space. However, we can use PCA as an alternative to visualize how your model classifies:

[31]:
```
# Reduce Dimensions to 2D
# Use PCA to compress features into 2 dimensions
pca = PCA(n_components=2); X_test_pca = pca.fit_transform(X_test_processed)  #␣
 ↪Use your preprocessed test data
# Create a meshgrid for decision boundary
x_min, x_max = X_test_pca[:, 0].min() - 1, X_test_pca[:, 0].max() + 1;
y_min, y_max = X_test_pca[:, 1].min() - 1, X_test_pca[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02), np.arange(y_min, y_max, 0.
 ↪02))

# Predict on meshgrid points
mesh_points = np.c_[xx.ravel(), yy.ravel()]; mesh_points_inverse = pca.
 ↪inverse_transform(mesh_points)  # Project back to original space
Z = model.predict(mesh_points_inverse).reshape(xx.shape)
```

```
6307/6307 [==============================] - 2s 364us/step
```

[32]:
```
# Create a wider figure with adjusted spacing
plt.figure(figsize=(9, 3))
# PCA Components
plt.subplot(1, 2, 1)  # 1 row, 2 columns, first plot
plt.contourf(xx, yy, Z, alpha=0.3, cmap='coolwarm'); plt.scatter(X_test_pca[:,␣
 ↪0], X_test_pca[:, 1], c=y_test, cmap='coolwarm', edgecolors='k', s=50)
plt.xlabel('PCA Component 1', fontsize=10); plt.ylabel('PCA Component 2',␣
 ↪fontsize=10); plt.title('PCA Projection with True Labels', fontsize=11)
cbar1 = plt.colorbar(label='Approval (1) / Denial (0)'); cbar1.ax.
 ↪tick_params(labelsize=9)  # Adjust colorbar font size

# Decision Boundary
plt.subplot(1, 2, 2)  # 1 row, 2 columns, second plot
plt.contourf(xx, yy, Z, alpha=0.3, levels=[0, 0.5, 1], cmap='coolwarm')
plt.scatter(X_test_pca[:, 0], X_test_pca[:, 1], c=y_test, cmap='coolwarm',␣
 ↪edgecolor='k', s=50)
plt.xlabel('PCA Component 1', fontsize=10); plt.ylabel('PCA Component 2',␣
 ↪fontsize=10)
plt.title('Model Decision Boundary', fontsize=11); cbar2 = plt.
 ↪colorbar(label='True Approval (1) / Denial (0)')
cbar2.ax.tick_params(labelsize=9)  # Adjust colorbar font size

# Add space between subplots
plt.tight_layout(w_pad=3.0)  # Increase horizontal padding (default is 1.0)
plt.show()
```



Key Interpretations Decision Regions: Red Zone: Model predicts approval (prob $>$ 0.5). Blue Zone: Model predicts denial (prob 0.5).

Points: Colored by true labels (red = approved, blue = denied). Misclassified points appear in the "wrong" colored region.

PCA Limitations: This is a 2D approximation—real decisions use all features. Overlapping clusters are expected (your AUC=0.943 confirms good separation).

What PCA Components represent
Principal Component 1 (PC1):
The direction in your original feature space where the data varies the most. Likely dominated by features with the strongest impact on predictions, likely A8 or A9. If A8 has high variance and strongly predicts approvals, PC1 will align closely with it.
Points further to the right have higher values in the original features that dominate PC1, points to the left have lower values in those features.
Principal Component 2 (PC2): The direction with the next highest variance, orthogonal (perpendicular) to PC1. Captures secondary patterns not explained by PC1, likely A10 or A6. Points higher up have higher values in features that dominate PC2, points lower down have lower values.
PCA Helps spot broad trends (e.g., clusters of approvals/denials).
Not for Exact Decisions: Always validate with original features.