Professor : Dr.Bagher Babaali

Student : Ahmad Asadi

Student Number : 610303080

Homework 5 : Machine Learning

Question 2

Step 1: Install Libraries

```
!pip install matplotlib seaborn
```

Step 2: Import Libraries

```
import pandas as pd
import numpy as np

import math

from sklearn.feature_selection import mutual_info_classif
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report

import matplotlib.pyplot as plt
import seaborn as sns
```

Pandas : Working with data frames and loading datasets into these frames

Numpy : have predefined functions for working with 2D arrays

math : using some calculation for calculating distance

mutual_info_classif : for getting the MI Score for feature selecting

train_test_split : for splitting the dataset into train and validation

SVC : the core for training the SVM model

GridSearchCV : for training the optimized hyperparameters

accuracy_score : for calculating accuracy score

classification_report : for balance accuracy between classes

matplotlib : showing charts

seaborn : showing charts more beautiful

Step 3: Get raw Dataset

```
train_set_url = "https://raw.githubusercontent.com/AsadiAhmad/Loan-Prediction-SVM/refs/heads/main/Dataset/train.csv"
test_set_url = "https://raw.githubusercontent.com/AsadiAhmad/Loan-Prediction-SVM/refs/heads/main/Dataset/test.csv"

pd.set_option('display.max_rows', None)

train_set = pd.read_csv(train_set_url)
test_set = pd.read_csv(test_set_url)
```

We are getting dataset from the csv file in the github repository

Step 4: Preprocessing

Transform discrete columns to numeric data

```
def transform_binary_columns(dataset):
    married_mapping = {'married': 1, 'single': 0}
    house_mapping = {'owned': 1, 'rented': 0.5, 'norent_noown': 0}
    car_mapping = {'yes': 1, 'no': 0}

    dataset['Married/Single'] = dataset['Married/Single'].map(married_mapping)
    dataset['House_Ownership'] = dataset['House_Ownership'].map(house_mapping)
    dataset['Car_Ownership'] = dataset['Car_Ownership'].map(car_mapping)

    return dataset
train_set_transformed = transform_binary_columns(train_set)
test_set_transformed = transform_binary_columns(test_set)
```

we should convert discrete string columns to numeric data

One Hut encoding

```
def one_hot_encoding(dataset, columns):
    new_columns = []
    for col in columns:
        unique_values = dataset[col].unique()
        for value in unique_values:
            new_col_name = f"{col}-{value}"
            new_columns.append((new_col_name, (dataset[col] ==
value).astype(int)))
    new_columns_df = pd.DataFrame(dict(new_columns))
    dataset = pd.concat([dataset, new_columns_df], axis=1)

    return dataset.drop(columns, axis=1)
```

```
train_set_one_hut = one_hot_encoding(train_set_transformed, ['Profession',
'CITY', 'STATE'])
test_set_one_hut = one_hot_encoding(test_set_transformed, ['Profession',
'CITY', 'STATE'])
```

some columns have too different values which we cant sort them like city or state then we should use one hut encoding

Normalizing min max

```
def min_max_normalize(train_set, test_set, columns):
    min_values = train_set[columns].min()
    max_values = train_set[columns].max()

    train_set_normalized = train_set.copy()
    train_set_normalized[columns] = (train_set[columns] - min_values) /
(max_values - min_values)

    test_set_normalized = test_set.copy()
    test_set_normalized[columns] = (test_set[columns] - min_values) /
(max_values - min_values)

    return train_set_normalized, test_set_normalized
```

```
columns_to_normalize = ["Income", "Age", "Experience", "CURRENT_JOB_YRS",
"CURRENT_HOUSE_YRS"]
```

```
train_set_normal, test_set_normal = min_max_normalize(train_set_one_hut,
test_set_one_hut, columns_to_normalize)
```

we should normalizing because have different range of values in the dataset

Remove unrelated features

```
train_set = train_set_normal.drop(["Id"], axis=1)
test_set = test_set_normal.drop(["Id"], axis=1)
```

Move Target col to the End

```
train_set = train_set[[col for col in train_set.columns if col != "Risk_Flag"]
+ ["Risk_Flag"]]
```

Step 5: Feature Selection with MI Score

```
train_set_copy = train_set.copy()
features_train_selection = train_set_copy.drop(columns=['Risk_Flag'])
target_train_selection = train_set_copy['Risk_Flag']

mutual_info = mutual_info_classif(features_train_selection,
target_train_selection)

mi_df = pd.DataFrame({
    'Feature': features_train_selection.columns,
    'Mutual Information': mutual_info
})

mi_df = mi_df.sort_values(by='Mutual Information', ascending=False)
```

```
selected_features = mi_df[mi_df['Mutual Information'] > 0.002]
selected_features
```

selecting best features of dataset

	Feature	Mutual Information
0	Income	0.159516
4	House_Ownership	0.024205
5	Car_Ownership	0.009337
7	CURRENT_HOUSE_YRS	0.007655
2	Experience	0.003407
380	STATE-Uttar_Pradesh	0.003329
6	CURRENT_JOB_YRS	0.003165
247	CITY-Nadiad	0.002270
167	CITY-Jodhpur	0.002180
1	Age	0.002119
381	STATE-Maharashtra	0.002076
80	CITY-Bhatpara	0.002075

Most important features we selected here are Income, House Ownership Car Ownership

Step 6: Split Dataset into Train and validation

```
train_set, validation_set = train_test_split(train_set, test_size=0.2,
random_state=42)
train_set = train_set.reset_index(drop=True)
validation_set = validation_set.reset_index(drop=True)
```

Step 7: Split Selected Features and Target Column

```
features_train = train_set[selected_features_list]
target_train = train_set['Risk_Flag']

features_validation = validation_set[selected_features_list]
target_validation = validation_set['Risk_Flag']
```

Step 8: Undersampling Train set

```
minority_class = features_train[target_train == 1]
majority_class = features_train[target_train == 0]

minority_class_size = len(minority_class)

majority_class_undersampled = majority_class.sample(n=minority_class_size,
random_state=42)

features_train_sample = pd.concat([minority_class,
majority_class_undersampled])
target_train_sample = pd.concat([target_train[target_train == 1],
target_train[target_train == 0].sample(n=minority_class_size,
random_state=42)])

features_train_sample = features_train_sample.sample(frac=1,
random_state=42).reset_index(drop=True)
target_train_sample = target_train_sample.sample(frac=1,
random_state=42).reset_index(drop=True)
```

in this step we just remove targets with value 0 to having same rows as target 1 when we running a model with very different scale of values it would forget the rows with low values so we scale the dataset

Step 9: Train SVM model with Selected Features

Linear Kernel

```
model_linear = SVC(kernel='linear', class_weight='balanced')
model_linear.fit(features_train_sample, target_train_sample)
linear_predicted = model_linear.predict(features_validation)

print("Accuracy:", accuracy_score(target_validation, linear_predicted))
print("Classification Report:\n", classification_report(target_validation,
linear_predicted))
```

```
Accuracy: 0.4136656746031746
Classification Report:
              precision    recall  f1-score   support

           0       0.89      0.38      0.53     35343
           1       0.13      0.68      0.22      4977

    accuracy                           0.41     40320
   macro avg       0.51      0.53      0.38     40320
weighted avg       0.80      0.41      0.49     40320
```

The linear kernel is the simplest type of kernel in SVM. It is used when the data is linearly separable This kernel computes the dot product of the input vectors.

Mathematical Formula:

For two vectors x and x′, the linear kernel is given by:

$$K(x, x') = x^T x' + c$$

Where:

- $x^T x'$ is the dot product of the vectors,
- c is a constant (usually 0).

When to Use:

- When the data is **linearly separable** (or approximately so),
- Works well when there are fewer features and no complex decision boundaries.

Poly Kernel

```
model_poly = SVC(kernel='poly', class_weight='balanced')
model_poly.fit(features_train_sample, target_train_sample)
poly_predicted = model_poly.predict(features_validation)

print("Accuracy:", accuracy_score(target_validation, poly_predicted))
print("Classification Report:\n", classification_report(target_validation,
poly_predicted))
```

```
Accuracy: 0.43278769841269843
Classification Report:
               precision    recall  f1-score   support

           0       0.90      0.39      0.55     35343
           1       0.14      0.70      0.23      4977

    accuracy                           0.43     40320
   macro avg       0.52      0.55      0.39     40320
weighted avg       0.81      0.43      0.51     40320
```

The polynomial kernel can capture nonlinear relationships between data points. It computes a higher-degree polynomial of the dot product of two vectors, allowing the SVM to fit a hyperplane that is more flexible and can represent more complex decision boundaries.

Mathematical Formula:

The polynomial kernel is given by:

$$K(x, x') = (x^T x' + c)^d$$

Where:

- d is the degree of the polynomial,
- c is a constant (typically 0 or 1).

When to Use:

When the data is nonlinearly separable but you still want to fit a polynomial decision boundary.

Useful for problems where you know the relationship between features is polynomial in nature

RBF Kernel

```
model_rbf = SVC(kernel='rbf', class_weight='balanced')
model_rbf.fit(features_train_sample, target_train_sample)
rbf_predicted = model_rbf.predict(features_validation)

print("Accuracy:", accuracy_score(target_validation, rbf_predicted))
print("Classification Report:\n", classification_report(target_validation,
rbf_predicted))
```

```
Accuracy: 0.4898313492063492
Classification Report:
               precision    recall  f1-score   support

           0       0.92      0.46      0.61     35343
           1       0.15      0.70      0.25      4977

    accuracy                           0.49     40320
   macro avg       0.53      0.58      0.43     40320
weighted avg       0.82      0.49      0.57     40320
```

The RBF kernel is one of the most commonly used kernels in SVMs and is particularly good for cases where the decision boundary is very complex or the data is not linearly separable. It is based on the distance between data points and maps the input features into an infinite-dimensional space, making it very flexible.

Mathematical Formula:

The RBF kernel is given by:

$$K(x, x') = \exp(\frac{||x - x'||^2}{2\sigma^2})$$

Where:

- $\| x-x' \|$ is the Euclidean distance between two vectors,
- σ is a parameter that controls the width of the Gaussian function (often related to the gamma parameter in SVM).

When to Use:

When the data is not linearly separable, and you need a flexible, complex decision boundary.

Commonly used in image classification, speech recognition, and many other applications where the decision boundary is complex.

Sigmoid Kernal

```
model_sigmoid = SVC(kernel='sigmoid', class_weight='balanced')
model_sigmoid.fit(features_train_sample, target_train_sample)
sigmoid_predicted = model_sigmoid.predict(features_validation)

print("Accuracy:", accuracy_score(target_validation, sigmoid_predicted))
print("Classification Report:\n", classification_report(target_validation,
sigmoid_predicted))
```

```
Accuracy: 0.5262400793650793
Classification Report:
               precision    recall  f1-score   support

           0       0.89      0.53      0.66     35343
           1       0.14      0.53      0.22      4977

    accuracy                           0.53     40320
   macro avg       0.51      0.53      0.44     40320
weighted avg       0.80      0.53      0.61     40320
```

The sigmoid kernel is based on the sigmoid activation function often used in neural networks. It can model nonlinear relationships, but it is not as widely used in practice for SVMs because it can behave unpredictably.

Mathematical Formula:

The sigmoid kernel is given by:

$$K(x, x') = \tanh(\alpha x^T x' + c)$$

Where:

- α is a scaling factor,
- c is a constant.

When to Use:

- It's sometimes used in neural networks and can be tried when you want a kernel function similar to the activation functions used in neural networks (sigmoid or tanh).

Step 10: Finding optimized hyperparameters

Initiate Param Grid

```
param_grid = {
    'C': [0.01, 0.1, 1, 10, 100]
}
```

Set low row dataset for find best hyper parameter

```
train_set_mini = train_set.sample(n=1000, random_state=42)
validation_set_mini = validation_set.sample(n=200, random_state=42)

features_train_mini = train_set_mini[selected_features_list]
target_train_mini = train_set_mini['Risk_Flag']

features_validation_mini = validation_set_mini[selected_features_list]
target_validation_mini = validation_set_mini['Risk_Flag']
```

Linear Kernel

```
model_linear = SVC(kernel='linear', gamma='scale')
grid_search_linear = GridSearchCV(model_linear, param_grid, cv=5,
scoring='f1', verbose=1)
grid_search_linear.fit(features_train_mini, target_train_mini)

print("Best C value:", grid_search_linear.best_params_['C'])
print("Best Training F-measure:", grid_search_linear.best_score_)

best_model_linear = grid_search_linear.best_estimator_
linear_predicted_mini = best_model_linear.predict(features_validation_mini)
linear_vaildation_accuracy_mini = accuracy_score(target_validation_mini,
linear_predicted_mini)
print("Best Validation accuracy", linear_vaildation_accuracy_mini)
```

```
Fitting 5 folds for each of 5 candidates, totalling 25 fits
Best C value: 0.01
Best Training accuracy: 0.0
Best Validation accuracy 0.91
```

```
model_linear = SVC(kernel='linear', C=grid_search_linear.best_params_['C'])
model_linear.fit(features_train_sample, target_train_sample)
linear_predicted = model_linear.predict(features_validation)

print("Accuracy:", accuracy_score(target_validation, linear_predicted))
print("Classification Report:\n", classification_report(target_validation,
linear_predicted))
```

```
Accuracy: 0.3595982142857143
Classification Report:
              precision    recall  f1-score   support

           0       0.89      0.31      0.46     35343
           1       0.13      0.72      0.22      4977

    accuracy                           0.36     40320
   macro avg       0.51      0.51      0.34     40320
weighted avg       0.79      0.36      0.43     40320
```

Poly Kernel

```
model_poly = SVC(kernel='poly', gamma='scale')
grid_search_poly = GridSearchCV(model_poly, param_grid, cv=5, scoring='f1',
verbose=1)
grid_search_poly.fit(features_train_mini, target_train_mini)

print("Best C value:", grid_search_poly.best_params_['C'])
print("Best F-measure:", grid_search_poly.best_score_)

best_model_poly = grid_search_poly.best_estimator_
poly_predicted_mini = best_model_poly.predict(features_validation_mini)
poly_vaildation_accuracy_mini = accuracy_score(target_validation_mini,
poly_predicted_mini)
print("Best Validation accuracy", poly_vaildation_accuracy_mini)
```

```
Fitting 5 folds for each of 5 candidates, totalling 25 fits
Best C value: 100
Best accuracy: 0.12913389242336612
Best Validation accuracy 0.875
```

```
model_poly = SVC(kernel='poly', C=grid_search_poly.best_params_['C'])
model_poly.fit(features_train_sample, target_train_sample)
poly_predicted = model_poly.predict(features_validation)

print("Accuracy:", accuracy_score(target_validation, poly_predicted))
print("Classification Report:\n", classification_report(target_validation,
poly_predicted))
```

```
Accuracy: 0.4421130952380952
Classification Report:
              precision    recall  f1-score   support

           0       0.90      0.41      0.56     35343
           1       0.14      0.68      0.23      4977

    accuracy                           0.44     40320
   macro avg       0.52      0.55      0.40     40320
weighted avg       0.81      0.44      0.52     40320
```

RBF Kernel

```
model_rbf = SVC(kernel='rbf', gamma='scale')
grid_search_rbf = GridSearchCV(model_rbf, param_grid, cv=5, scoring='f1',
verbose=1)
grid_search_rbf.fit(features_train_mini, target_train_mini)

print("Best C value:", grid_search_rbf.best_params_['C'])
print("Best Training F-measure:", grid_search_rbf.best_score_)

best_model_rbf = grid_search_rbf.best_estimator_
rbf_predicted_mini = best_model_rbf.predict(features_validation_mini)
rbf_vaildation_accuracy_mini = accuracy_score(target_validation_mini,
rbf_predicted_mini)
print("Best Validation accuracy", rbf_vaildation_accuracy_mini)
```

```
Fitting 5 folds for each of 5 candidates, totalling 25 fits
Best C value: 100
Best Training F-measure: 0.15118206305098467
Best Validation accuracy 0.845
```

```
model_rbf = SVC(kernel='rbf', C=grid_search_rbf.best_params_['C'])
model_rbf.fit(features_train_sample, target_train_sample)
rbf_predicted = model_rbf.predict(features_validation)

print("Accuracy:", accuracy_score(target_validation, rbf_predicted))
print("Classification Report:\n", classification_report(target_validation,
rbf_predicted))
```

```
Accuracy: 0.5763392857142857
Classification Report:
              precision    recall  f1-score   support

           0       0.93      0.56      0.70     35343
           1       0.18      0.68      0.28      4977

    accuracy                           0.58     40320
   macro avg       0.55      0.62      0.49     40320
weighted avg       0.83      0.58      0.65     40320
```

Sigmoid Kernal

```
model_sigmoid = SVC(kernel='sigmoid', gamma='scale')
grid_search_sigmoid = GridSearchCV(model_sigmoid, param_grid, cv=5,
scoring='f1', verbose=1)
grid_search_sigmoid.fit(features_train_mini, target_train_mini)

print("Best C value:", grid_search_sigmoid.best_params_['C'])
print("Best Training F-measure:", grid_search_sigmoid.best_score_)

best_model_sigmoid = grid_search_sigmoid.best_estimator_
sigmoid_predicted_mini = best_model_sigmoid.predict(features_validation_mini)
sigmoid_vaildation_accuracy_mini = accuracy_score(target_validation_mini,
sigmoid_predicted_mini)
print("Best Validation accuracy", sigmoid_vaildation_accuracy_mini)
```

```
Fitting 5 folds for each of 5 candidates, totalling 25 fits
Best C value: 10
Best Training F-measure: 0.09728345411627809
Best Validation accuracy 0.83
```

```
model_sigmoid = SVC(kernel='sigmoid', C=grid_search_sigmoid.best_params_['C'])
model_sigmoid.fit(features_train_sample, target_train_sample)
sigmoid_predicted = model_sigmoid.predict(features_validation)

print("Accuracy:", accuracy_score(target_validation, sigmoid_predicted))
print("Classification Report:\n", classification_report(target_validation,
sigmoid_predicted))
```

```
Accuracy: 0.5198164682539682
Classification Report:
              precision    recall  f1-score   support

           0       0.88      0.52      0.65     35343
           1       0.13      0.52      0.21      4977

    accuracy                           0.52     40320
   macro avg       0.51      0.52      0.43     40320
weighted avg       0.79      0.52      0.60     40320
```

Step 11: Calculate Confusion Matrix

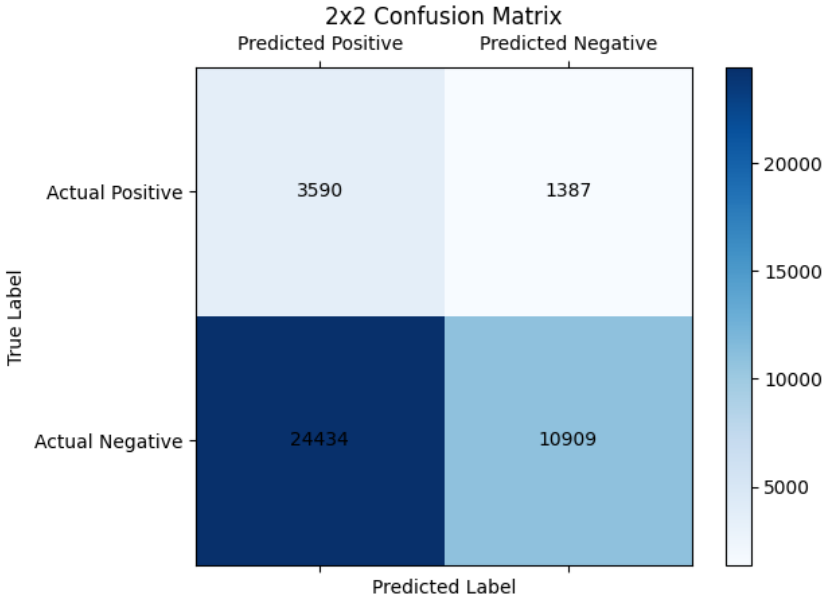
Core functions

```
def calculate_confusion_matrix(target_list, predicted_list):
    target_array = np.array(target_list)
    predicted_array = np.array(predicted_list)

    if target_array.shape[0] != predicted_array.shape[0]:
        raise ValueError("target_list and predicted_list must have the same
length.")

    TP = ((target_array == 1) & (predicted_array == 1)).sum()
    FN = ((target_array == 1) & (predicted_array == 0)).sum()
    FP = ((target_array == 0) & (predicted_array == 1)).sum()
    TN = ((target_array == 0) & (predicted_array == 0)).sum()

    return TP, FN, FP, TN
```

```
def show_confusion_matrix(tp, fn, fp, tn):
    conf_matrix = np.array([[tp, fn], [fp, tn]])
    fig, ax = plt.subplots()

    cax = ax.matshow(conf_matrix, cmap='Blues')

    plt.colorbar(cax)

    for (i, j), val in np.ndenumerate(conf_matrix):
        ax.text(j, i, f'{val}', ha='center', va='center', color='black')

    ax.set_xticks([0, 1])
    ax.set_yticks([0, 1])

    ax.set_xticklabels(['Predicted Positive', 'Predicted Negative'])
    ax.set_yticklabels(['Actual Positive', 'Actual Negative'])

    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
    plt.title('2x2 Confusion Matrix')

    plt.show()
```

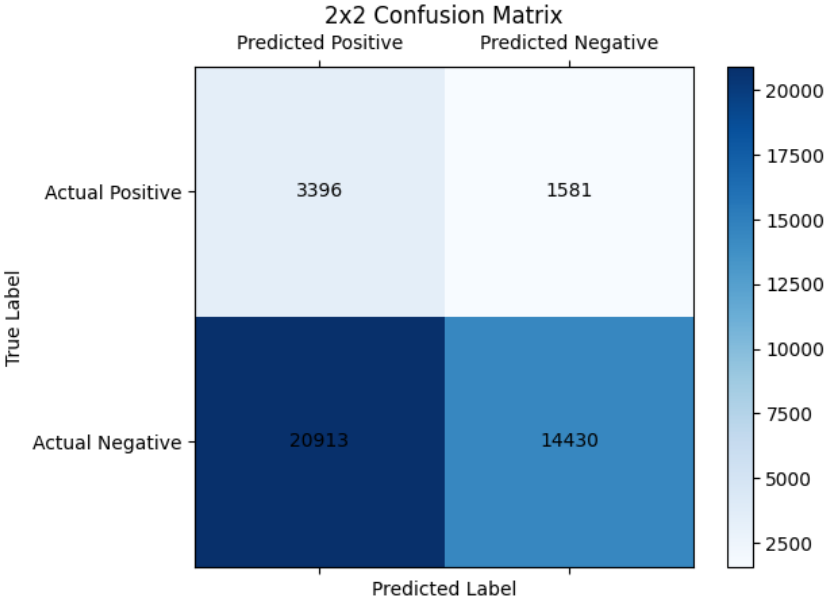
Linear Kernel

```
tp_linear, fn_linear, fp_linear, tn_linear =
calculate_confusion_matrix(target_validation, linear_predicted)
show_confusion_matrix(tp_linear, fn_linear, fp_linear, tn_linear)
```



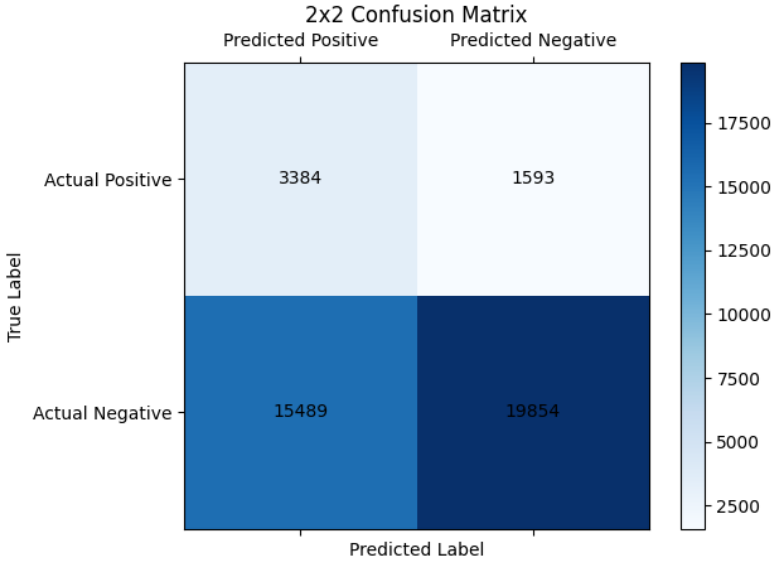
Poly Kernel

```
tp_poly, fn_poly, fp_poly, tn_poly =
calculate_confusion_matrix(target_validation, poly_predicted)
show_confusion_matrix(tp_poly, fn_poly, fp_poly, tn_poly)
```



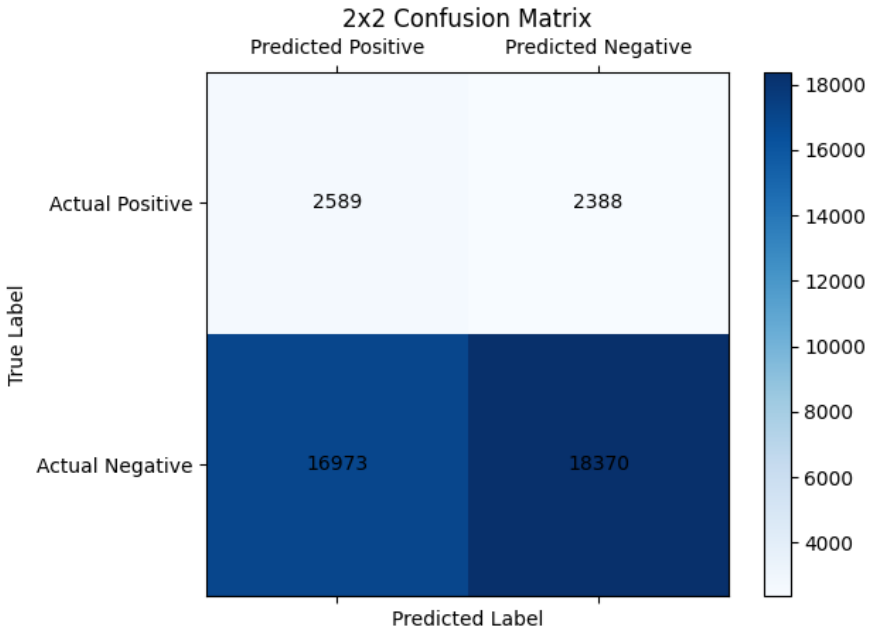
RBF Kernel

```
tp_rbf, fn_rbf, fp_rbf, tn_rbf = calculate_confusion_matrix(target_validation,
rbf_predicted)
show_confusion_matrix(tp_rbf, fn_rbf, fp_rbf, tn_rbf)
```



Sigmoid Kernal

```
tp_sigmoid, fn_sigmoid, fp_sigmoid, tn_sigmoid =
calculate_confusion_matrix(target_validation, sigmoid_predicted)
show_confusion_matrix(tp_sigmoid, fn_sigmoid, fp_sigmoid, tn_sigmoid)
```



Step 12: Calculate Measurements

Core function

```
def calculate_measurements(tp, fn, fp, tn):
    precision = tp / (tp + fp) if tp + fp != 0 else 0
    recall = tp / (tp + fn) if tp + fn != 0 else 0
    accuracy = (tp + tn) / (tp + tn + fp + fn) if tp + tn + fp + fn != 0 else
0
    f_measure = 2 * (precision * recall) / (precision + recall) if precision +
recall != 0 else 0

    precision *= 100
    recall *= 100
    accuracy *= 100
    f_measure *= 100

    print(f"Precision: {precision:.2f}%")
    print(f"Recall: {recall:.2f}%")
    print(f"Accuracy: {accuracy:.2f}%")
    print(f"F-Measure (F1 Score): {f_measure:.2f}%")
    return precision, recall, accuracy, f_measure
```

Linear Kernel

```
calculate_measurements(tp_linear, fn_linear, fp_linear, tn_linear)
```

```
Precision: 12.81%
Recall: 72.13%
Accuracy: 35.96%
F-Measure (F1 Score): 21.76%
(12.810448187268056, 72.1318063090215, 35.95982142857143, 21.756916457077057)
```

Poly Kernel

```
calculate_measurements(tp_poly, fn_poly, fp_poly, tn_poly)
```

```
Precision: 13.97%
Recall: 68.23%
Accuracy: 44.21%
F-Measure (F1 Score): 23.19%
(13.970134518079725, 68.23387582881254, 44.211309523809526, 23.191968858840404)
```

RBF Kernel

```
calculate_measurements(tp_rbf, fn_rbf, fp_rbf, tn_rbf)
```

```
Precision: 17.93%
Recall: 67.99%
Accuracy: 57.63%
F-Measure (F1 Score): 28.38%
(17.93037672865999, 67.99276672694394, 57.63392857142857, 28.37735849056604)
```

Sigmoid Kernal

```
calculate_measurements(tp_sigmoid, fn_sigmoid, fp_sigmoid, tn_sigmoid)
```

```
Precision: 13.23%
Recall: 52.02%
Accuracy: 51.98%
F-Measure (F1 Score): 21.10%
(13.234843063081483, 52.01928872814949, 51.98164682539682, 21.10110436448103)
```

Step 13: Calculate Test set with best model

```
features_test = test_set[selected_features_list]

best_model = model_rbf

predicted = best_model.predict(features_test)
test_set['Risk_Flag'] = predicted
```

Step 14: Export the predicted EXCEL

```
test_set.to_csv('predictions1.csv', index=False)
```

Result :

First running model give us a good accuracy but when try to find the best hyper parameter with low row dataset it gives us better accuracy about 10 percent! 😊 and final best kernel is RBF with accuracy 57% Percision 18%

in this Homework we have a lot of targets with value 1 ☹ and we should balance the dataset first in preprocessing section and finaly instead of increasing the value 1 rows I decide to remove value 0 rows because it costs a lot of time for training.

Here is more info about the project in the Github repo!

https://github.com/AsadiAhmad/Loan-Prediction-SVM