Support Vector Machines

In [ ]:
```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.svm import LinearSVC, SVC
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay, f1_score, pre
from tqdm import tqdm
```

First, load in our dataset.

In [ ]:
```
df = pd.read_csv("./data/nan_removed_cleaned_data.csv")
df.columns
```

Out[ ]:
```
Index(['budget', 'genres', 'id', 'imdb_id', 'original_title', 'overview',
       'popularity', 'poster_path', 'release_date', 'revenue', 'runtime',
       'spoken_languages', 'tagline', 'title', 'vote_average', 'vote_count',
       'log_popularity', 'title_length', 'num_languages', 'num_genres',
       'imdb_rating', 'imdb_budget', 'imdb_revenue', 'budget_currency',
       'revenue_currency', 'converted_budget', 'converted_revenue',
       'combined_budget', 'combined_revenue'],
      dtype='object')
```

SVMs are a supervised learning process for binary classification. So let's classify our data on a binary variable.

Just as we did with Naive Bayes, we can attempt to classify whether our film will be a box-office flop or success.

But this time, we can use most the data available to us instead of just a classifier based on the text description of the movie.

Let's set up the data:

In [ ]:
```
print(df.shape[0])

# First, convert our datetime objects to Unix timestamps
df['release_unix'] = pd.to_datetime(df['release_date'])
df['release_unix'] = df.release_unix.values.astype(np.int64) // 10 ** 9

# Next, only pick out any relevant columns
df = df[['runtime', 'vote_count', 'log_popularity', 'title_length', 'num_languages'
         'combined_revenue', 'release_unix']]

# Next, remove any missing data
df.dropna(inplace=True)

df.shape[0]
```

```
16012
```

Out[ ]:

```
13894
```

As shown, we only lost 2118 rows after removing missing data. We're left with 13894 rows of data!

We can use the same method as in Naive Bayes to create a new column that holds whether a film is a box office success or flop (binary).

In [ ]:

```python
def box_office_success(row):
    if row['combined_budget'] < row['combined_revenue']:
        return 1
    else:
        return 0

df['labels'] = df.apply(box_office_success, axis=1)

df = df.drop(['combined_budget', 'combined_revenue'], axis=1)

display(df.head(10))

display(df['labels'].value_counts())
```

	runtime	vote_count	log_popularity	title_length	num_languages	num_genres	imdb_rating	re
0	80	21	1.209259	32	5	1	8.1	1
1	100	17305	4.680380	12	1	2	8.2	1(
2	122	10783	3.274462	15	1	1	8.4	'
3	141	1506	2.635121	18	1	2	7.9	'
4	87	225	2.127994	8	2	3	5.3	1
5	126	9451	3.880470	17	3	5	7.6	;
6	106	398	2.471653	18	1	2	7.4	1(
7	91	88	2.077690	18	1	1	7.6	-
8	143	18370	4.407950	54	1	3	8.1	1(
9	111	15437	3.506308	17	3	2	8.2	1(

```
0    11684
1     2210
Name: labels, dtype: int64
```

Cool! As we can see, there are 2210 box office successes, and 11684 flops. Although we have a data imbalance, we can still apply SVM to our problem.

We'll want to keep the same proportion of class values in our training and testing set. Let's split our data:

In [ ]:

```
# First, separate class labels
df_success = df[df['labels'] == 1]
df_fail = df[df['labels'] == 0]

# Next, just separate into predictors and response
x_success = df_success.drop('labels', axis=1)
y_success = df_success['labels']
x_fail = df_fail.drop('labels', axis=1)
y_fail = df_fail['labels']

# Next, take out specific class labels
x_success_train, x_success_test, y_success_train, y_success_test = train_test_split
x_fail_train, x_fail_test, y_fail_train, y_fail_test = train_test_split(x_fail, y_f

# Concatenate everything
x_train = pd.concat([x_success_train, x_fail_train])
x_test = pd.concat([x_success_test, x_fail_test])
y_train = pd.concat([y_success_train, y_fail_train])
y_test = pd.concat([y_success_test, y_fail_test])

# # And shuffle
# x_train = x_train.sample(frac=1, random_state=1)
# x_test = x_test.sample(frac=1, random_state=1)
# y_train = y_train.sample(frac=1, random_state=1612)
# y_test = y_test.sample(frac=1, random_state=1612)
```

In [ ]:

```
print(len(x_test))
print(len(x_train))
```

```
2779
11115
```

Great, now we have our separated data!

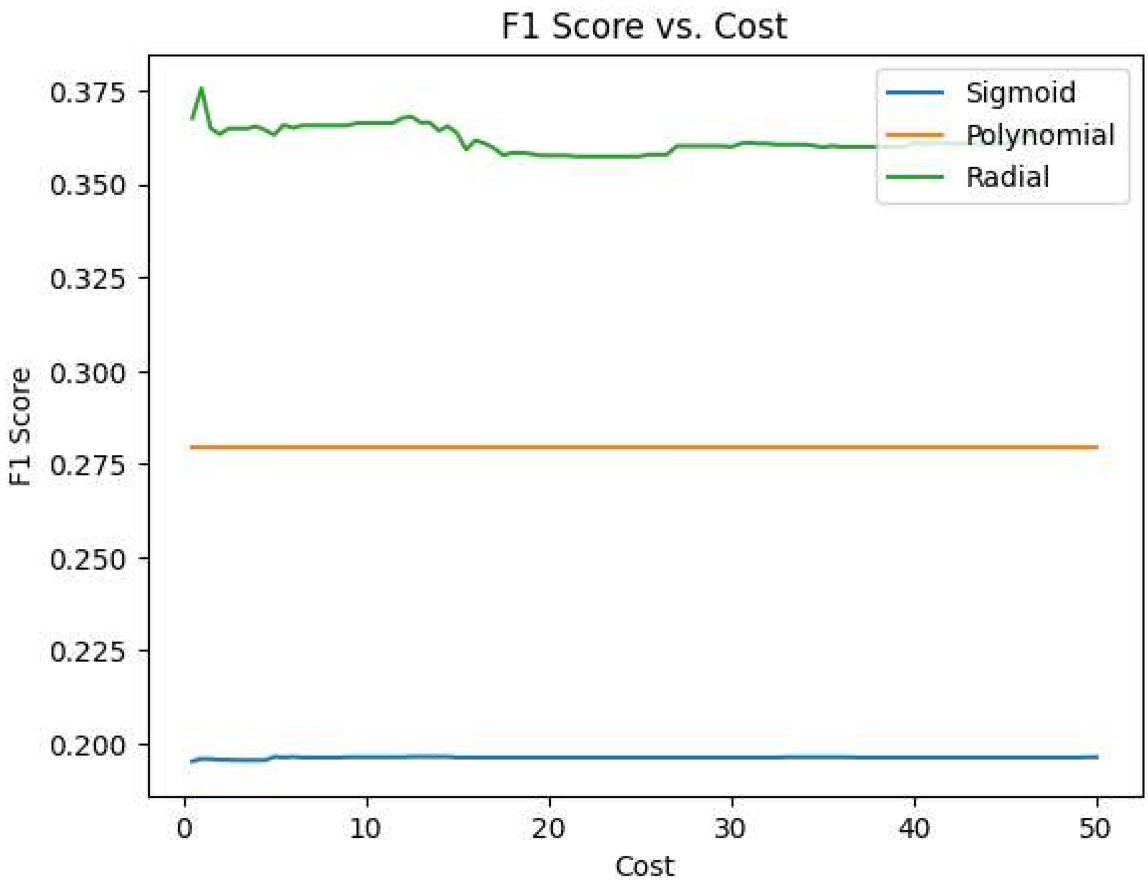
Let's run SVM on this:

In [ ]:

```
# Lists of f1 scores
sigmoid_f1s = []
poly_f1s = []
radial_f1s = []

# Range of costs to iterate over
costs = np.arange(0.5, 50.5, 0.5)

# Loop through a bunch of costs to determine the best one for each kernel
for cost in tqdm(costs):
    # Set up the SVMs - note the "balanced" class_weight parameter to alleviate iss
    SVM_sigmoid = SVC(C=cost, kernel='sigmoid', class_weight='balanced')
    SVM_poly = SVC(C=cost, kernel='poly', class_weight='balanced')
    SVM_radial = SVC(C=cost, kernel='rbf', class_weight='balanced')

    # Fit each to the train data
    SVM_sigmoid.fit(x_train, y_train)
    SVM_poly.fit(x_train, y_train)
    SVM_radial.fit(x_train, y_train)
```

```
    # Predict each on the test data
    pred_sigmoid = SVM_sigmoid.predict(x_test)
    pred_poly = SVM_poly.predict(x_test)
    pred_radial = SVM_radial.predict(x_test)

    # Calculate the F1 for sigmoid
    sigmoid_f1 = f1_score(y_test, pred_sigmoid)

    # Calculate the F1 for poly
    poly_f1 = f1_score(y_test, pred_poly)

    # Calculate the F1 for radial
    radial_f1 = f1_score(y_test, pred_radial)

    # Append to arrays
    sigmoid_f1s.append(sigmoid_f1)
    poly_f1s.append(poly_f1)
    radial_f1s.append(radial_f1)

plt.figure()
plt.plot(costs, sigmoid_f1s, label='Sigmoid')
plt.plot(costs, poly_f1s, label='Polynomial')
plt.plot(costs, radial_f1s, label='Radial')
plt.legend()
plt.xlabel('Cost')
plt.ylabel('F1 Score')
plt.title('F1 Score vs. Cost')
plt.savefig('./imgs/svm_ims/cost_comparison.png')
plt.show()


print(f"Best cost for sigmoid kernel: {costs[np.argmax(sigmoid_f1s)]}, F1 = {np.max
print(f"Best cost for polynomial kernel: {costs[np.argmax(poly_f1s)]}, F1 = {np.max
print(f"Best cost for radial kernel: {costs[np.argmax(radial_f1s)]}, F1 = {np.max(r
```

```
100%|██████████| 100/100 [30:24<00:00, 18.25s/it]
```



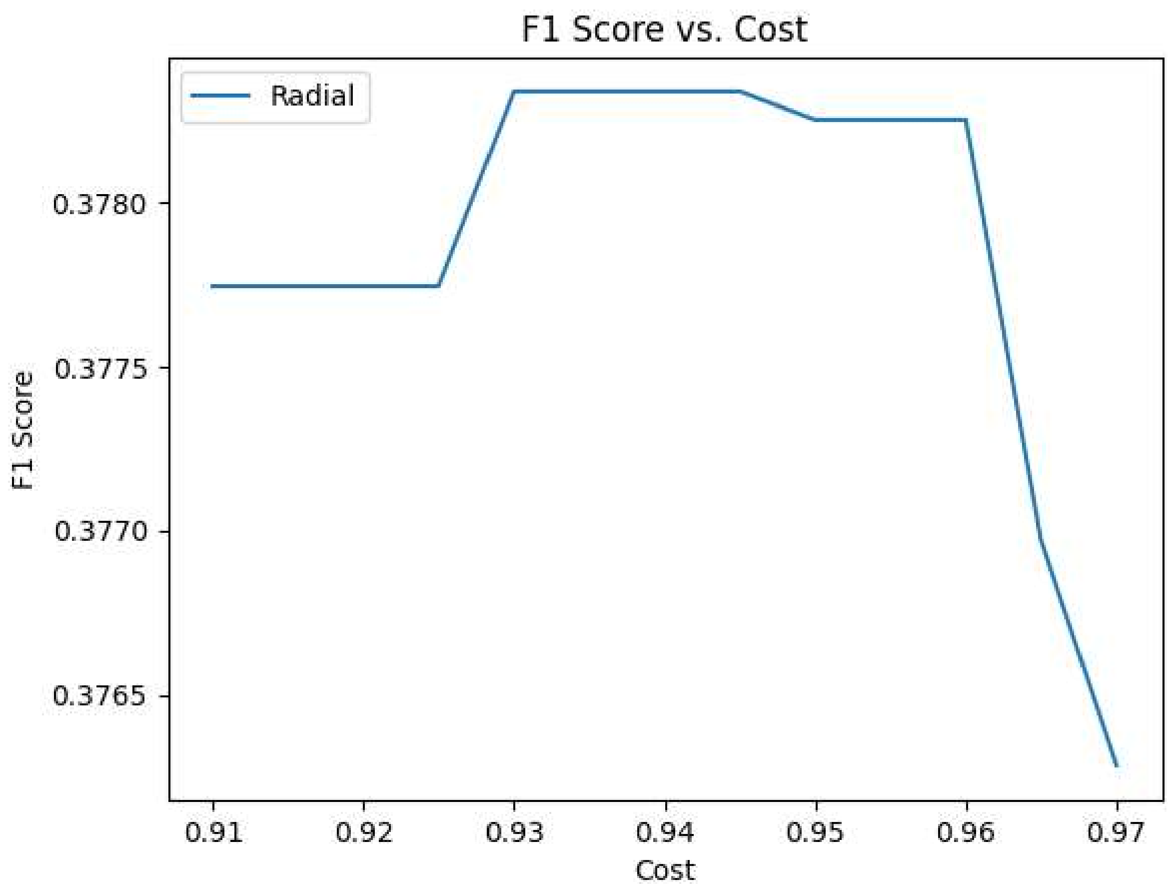

```
Best cost for sigmoid kernel: 5.0, F1 = 0.19641943734015344
Best cost for polynomial kernel: 0.5, F1 = 0.2793791574279379
Best cost for radial kernel: 1.0, F1 = 0.37569060773480667
```

In [ ]:

```
# Lists of f1 scores
new_radial_f1s = []

# Range of costs to iterate over
costs = np.arange(0.91, 0.975, 0.005)

# Loop through a bunch of costs to determine the best one for each kernel
for cost in tqdm(costs):
    # Set up the SVMs - note the "balanced" class_weight parameter to alleviate iss
    SVM_radial = SVC(C=cost, kernel='rbf', class_weight='balanced')

    # Fit each to the train data
    SVM_radial.fit(x_train, y_train)

    # Predict each on the test data
    pred_radial = SVM_radial.predict(x_test)

    # Calculate the F1 for radial
    radial_f1 = f1_score(y_test, pred_radial)

    # Append to arrays
    new_radial_f1s.append(radial_f1)

plt.figure()
plt.plot(costs, new_radial_f1s, label='Radial')
```

```
plt.legend()
plt.xlabel('Cost')
plt.ylabel('F1 Score')
plt.title('F1 Score vs. Cost')
plt.savefig('./imgs/svm_ims/cost_comparison_radial.png')
plt.show()

print(f"Best cost for radial kernel: {costs[np.argmax(new_radial_f1s)]}, F1 = {np.m
```

```
100%|██████████| 13/13 [01:42<00:00,  7.86s/it]
```




```
Best cost for radial kernel: 0.93, F1 = 0.37833594976452123
```

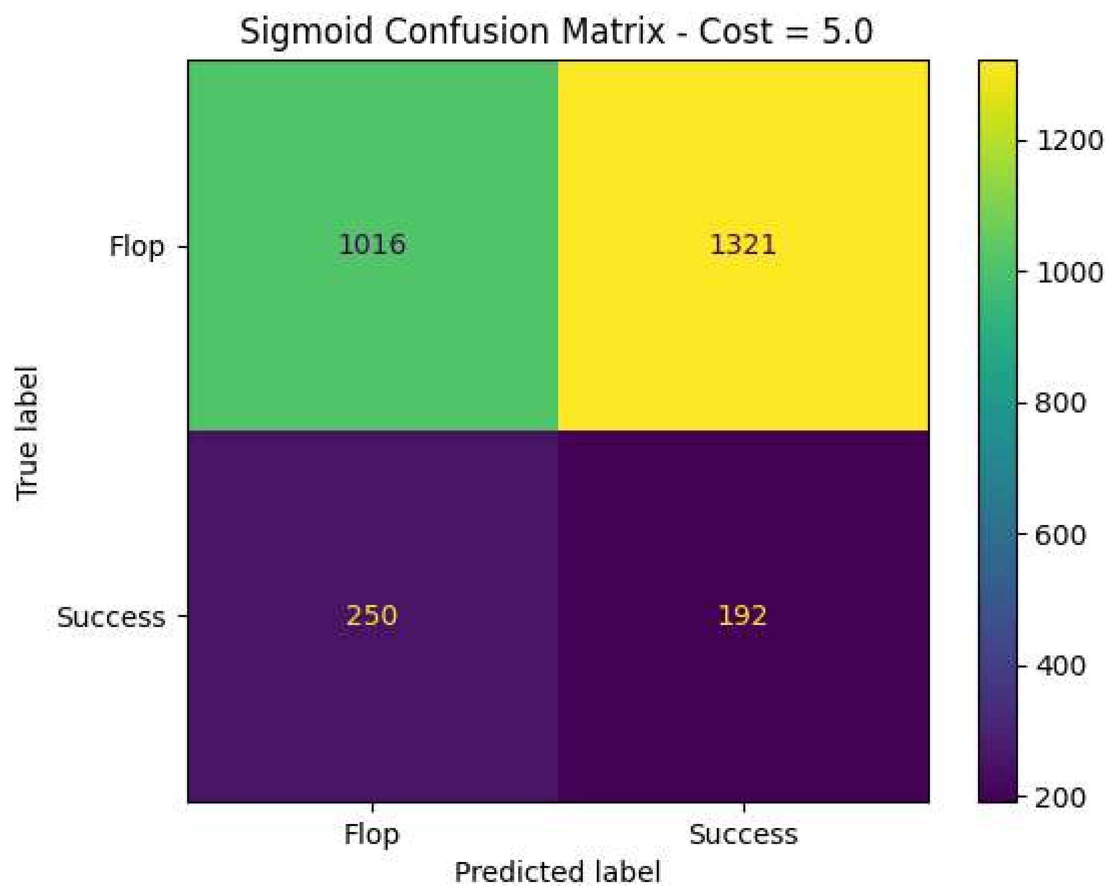
Great! Now we have (roughly) the *ideal* cost for each kernel. We can run the SVM on these specific models to evaluate our performance.

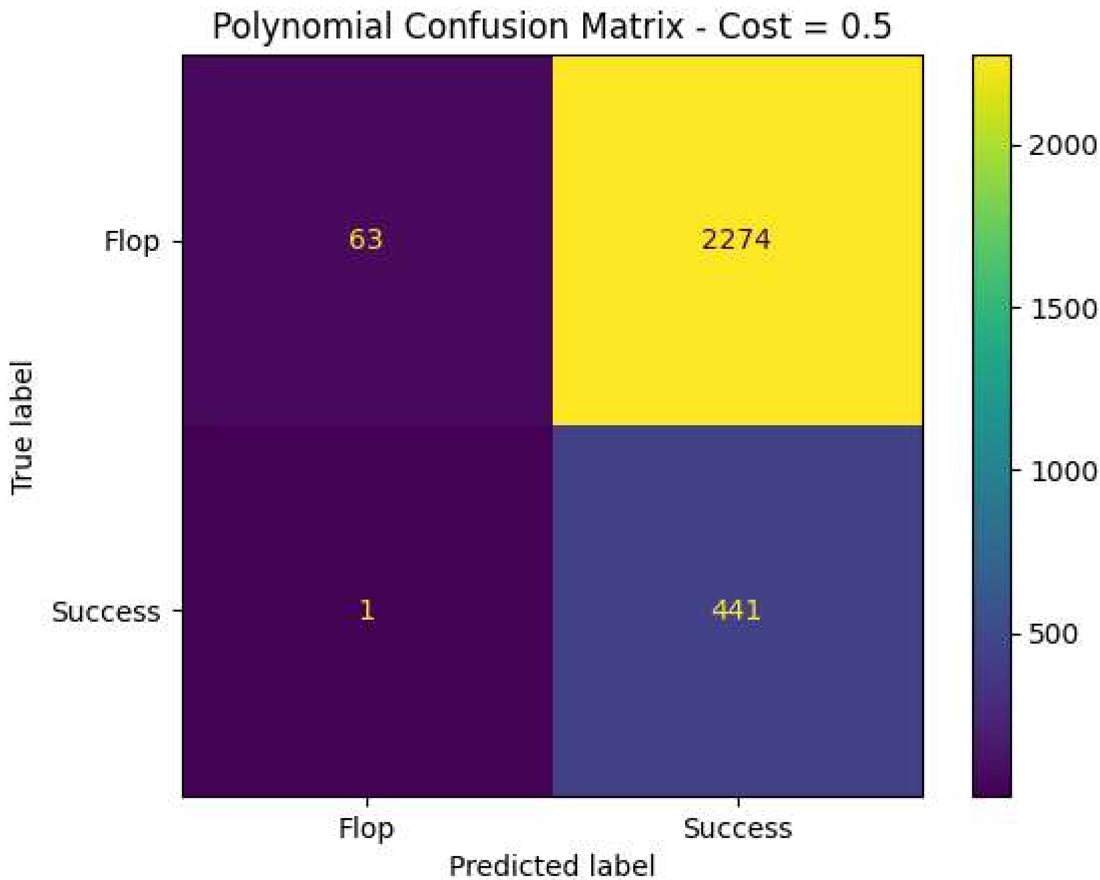
In [ ]:

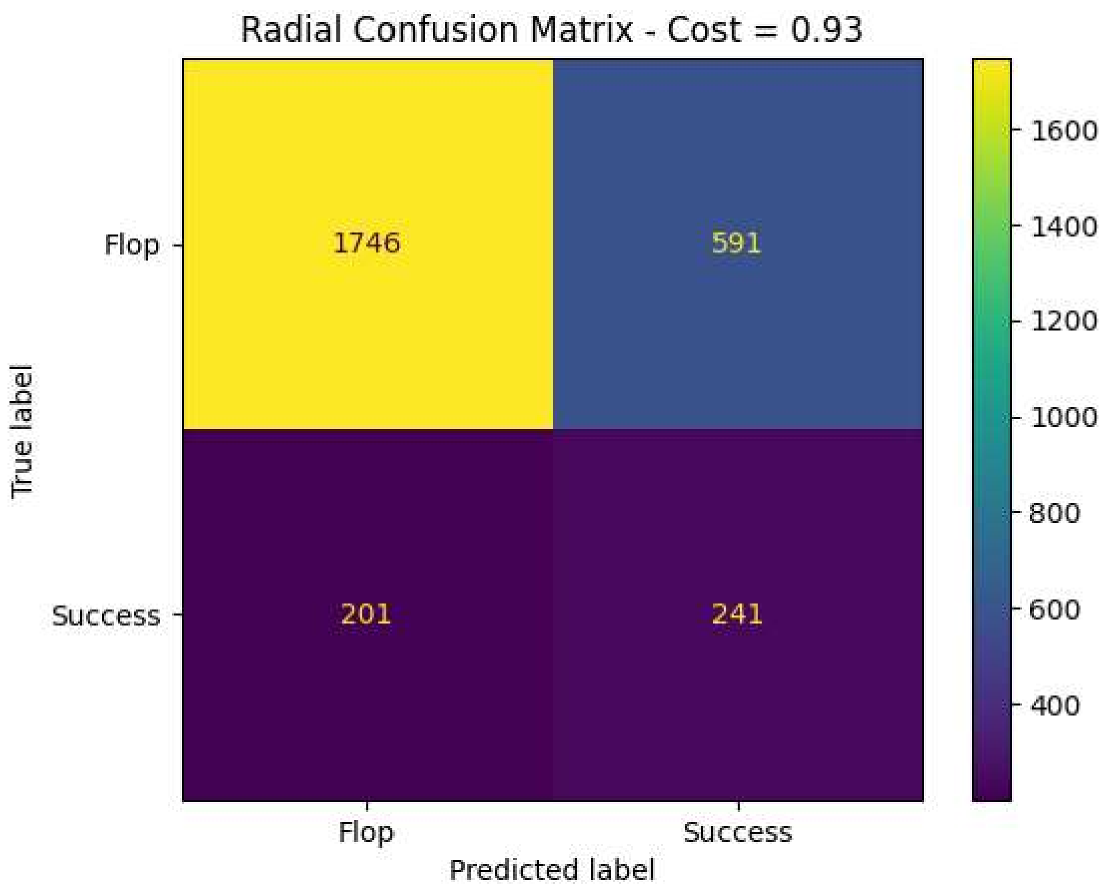
```
# Set up the SVMs
SVM_sigmoid = SVC(C=5.0, kernel='sigmoid', class_weight='balanced')
SVM_poly = SVC(C=0.5, kernel='poly', class_weight='balanced')
SVM_radial = SVC(C=0.93, kernel='rbf', class_weight='balanced')

# Fit each to the train data
SVM_sigmoid.fit(x_train, y_train)
SVM_poly.fit(x_train, y_train)
SVM_radial.fit(x_train, y_train)

# Predict each on the test data
pred_sigmoid = SVM_sigmoid.predict(x_test)
pred_poly = SVM_poly.predict(x_test)
```

```
pred_radial = SVM_radial.predict(x_test)

# Create confusion matrices
matrix_sigmoid = confusion_matrix(y_test, pred_sigmoid)
matrix_poly = confusion_matrix(y_test, pred_poly)
matrix_radial = confusion_matrix(y_test, pred_radial)

# Display conf. matrices
# Also print info statistics about model performance
disp = ConfusionMatrixDisplay(matrix_sigmoid, display_labels=['Flop', 'Success'])
disp.plot()
plt.title('Sigmoid Confusion Matrix - Cost = 5.0')
plt.savefig('./imgs/svm_ims/confusion_sigmoid.png')

print("Sigmoid Stats:")
sigmoid_prec, sigmoid_recall, _, sigmoid_support = precision_recall_fscore_support(
sigmoid_acc = accuracy_score(y_test, pred_sigmoid)
sigmoid_f1 = f1_score(y_test, pred_sigmoid)
print(f"Accuracy = {sigmoid_acc:.4f}")
print(f"Average Precision = {np.mean(sigmoid_prec):.4f}")
print(f"Average Recall = {np.mean(sigmoid_recall):.4f}")
print(f"F1 Score = {sigmoid_f1:.4f}")
print(f"Support = {sigmoid_support}\n")

disp = ConfusionMatrixDisplay(matrix_poly, display_labels=['Flop', 'Success'])
disp.plot()
plt.title('Polynomial Confusion Matrix - Cost = 0.5')
plt.savefig(f'./imgs/svm_ims/confusion_poly.png')

print("Polynomial Stats:")
poly_prec, poly_recall, _, poly_support = precision_recall_fscore_support(y_test, p
poly_acc = accuracy_score(y_test, pred_poly)
poly_f1 = f1_score(y_test, pred_poly)
print(f"Accuracy = {poly_acc:.4f}")
print(f"Average Precision = {np.mean(poly_prec):.4f}")
print(f"Average Recall = {np.mean(poly_recall):.4f}")
print(f"F1 Score = {poly_f1:.4f}")
print(f"Support = {poly_support}\n")

disp = ConfusionMatrixDisplay(matrix_radial, display_labels=['Flop', 'Success'])
disp.plot()
plt.title('Radial Confusion Matrix - Cost = 0.93')
plt.savefig(f'./imgs/svm_ims/confusion_radial.png')

print("Radial Stats:")
radial_prec, radial_recall, _, radial_support = precision_recall_fscore_support(y_t
radial_acc = accuracy_score(y_test, pred_radial)
radial_f1 = f1_score(y_test, pred_radial)
print(f"Accuracy = {radial_acc:.4f}")
print(f"Average Precision = {np.mean(radial_prec):.4f}")
print(f"Average Recall = {np.mean(radial_recall):.4f}")
print(f"F1 Score = {radial_f1:.4f}")
print(f"Support = {radial_support}")
```

```
Sigmoid Stats:
Accuracy = 0.4347
Average Precision = 0.4647
Average Recall = 0.4346
F1 Score = 0.1964
Support = [2337  442]

Polynomial Stats:
Accuracy = 0.1814
Average Precision = 0.5734
Average Recall = 0.5123
F1 Score = 0.2794
Support = [2337  442]

Radial Stats:
Accuracy = 0.7150
Average Precision = 0.5932
Average Recall = 0.6462
F1 Score = 0.3783
Support = [2337  442]
```




Polynomial Confusion Matrix - Cost = 0.5
Flop
Success
63
2274
1
441
True label
Predicted label
2000
1500
1000
500


Radial Confusion Matrix - Cost = 0.93
Flop
Success
1746
591
201
241
True label
Predicted label
1600
1400
1200
1000
800
600
400

As shown above, our Radial kernel is the most effective across **all of the** statistics, as also visible in the confusion matrix when compared to the others.

Going back to our Naive Bayes model from earlier, none of our SVM models perform as well. This is surprising, since the SVMs are fed more data and are not restricted only to a text description of the film. However, it is likely due in part to the strong imbalance of data, since 84% of our original data is box office flops, with only 16% being successes. Although the `class_weight="balanced"` parameter in the SVM construction assists with this, it's tough to completely avoid the issue.