CS-GY 6923 Machine Learning Assignment - 3

Name: Sai Harsha Varma Sangaraju

NetID: ss18851

In [1]:
```
# Importing the libraries
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
from imblearn.under_sampling import RandomUnderSampler
from sklearn.preprocessing import MinMaxScaler
from sklearn.svm import SVC
from sklearn.metrics import classification_report
from sklearn.model_selection import train_test_split
from sklearn.inspection import DecisionBoundaryDisplay
```

In [2]:
```
# Loading the dataset
df = pd.read_csv('/kaggle/input/kddcup99/kddcup.data.corrected', header=None

# Displaying the shape of the dataset
print(f"Dataset shape: {df.shape}")

# Displaying the first few rows
df.head()
```

```
Dataset shape: (4898431, 42)
```

Out[2]:

	0	1	2	3	4	5	6	7	8	9	...	32	33	34	35	36	37	38	39
0	0	tcp	http	SF	215	45076	0	0	0	0	...	0	0.0	0.0	0.00	0.0	0.0	0.0	0.0
1	0	tcp	http	SF	162	4528	0	0	0	0	...	1	1.0	0.0	1.00	0.0	0.0	0.0	0.0
2	0	tcp	http	SF	236	1228	0	0	0	0	...	2	1.0	0.0	0.50	0.0	0.0	0.0	0.0
3	0	tcp	http	SF	233	2032	0	0	0	0	...	3	1.0	0.0	0.33	0.0	0.0	0.0	0.0
4	0	tcp	http	SF	239	486	0	0	0	0	...	4	1.0	0.0	0.25	0.0	0.0	0.0	0.0

5 rows × 42 columns

Exploratory Data Analysis

In [3]:
```
# Adding the column names
column_names = [
    'duration', 'protocol_type', 'service', 'flag', 'src_bytes', 'dst_bytes'
    'land', 'wrong_fragment', 'urgent', 'hot', 'num_failed_logins', 'logged_
    'num_compromised', 'root_shell', 'su_attempted', 'num_root',
    'num_file_creations', 'num_shells', 'num_access_files', 'num_outbound_cm
    'is_host_login', 'is_guest_login', 'count', 'srv_count', 'serror_rate',
    'srv_serror_rate', 'rerror_rate', 'srv_rerror_rate', 'same_srv_rate',
    'diff_srv_rate', 'srv_diff_host_rate', 'dst_host_count',
    'dst_host_srv_count', 'dst_host_same_srv_rate', 'dst_host_diff_srv_rate'
    'dst_host_same_src_port_rate', 'dst_host_srv_diff_host_rate',
    'dst_host_serror_rate', 'dst_host_srv_serror_rate', 'dst_host_rerror_rat
    'dst_host_srv_rerror_rate', 'label'
]

# Loading the dataset with the correct column names
df = pd.read_csv('/kaggle/input/kddcup99/kddcup.data.corrected', header=None

# Displaying the first few rows to verify column names are correctly assigne
df.head()# Checking the distribution of the 'label' column (attack types)
df['label'].value_counts()
```

Out[3]:
```
label
smurf.              2807886
neptune.            1072017
normal.              972781
satan.                15892
ipsweep.              12481
portsweep.            10413
nmap.                  2316
back.                  2203
warezclient.           1020
teardrop.               979
pod.                    264
guess_passwd.            53
buffer_overflow.         30
land.                    21
warezmaster.             20
imap.                    12
rootkit.                 10
loadmodule.               9
ftp_write.                8
multihop.                 7
phf.                      4
perl.                     3
spy.                      2
Name: count, dtype: int64
```

In [4]:
```
# Checking the data types of the columns
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4898431 entries, 0 to 4898430
Data columns (total 42 columns):
 #   Column                       Dtype  
---  ------                       -----  
 0   duration                     int64  
 1   protocol_type                object 
 2   service                      object 
 3   flag                         object 
 4   src_bytes                    int64  
 5   dst_bytes                    int64  
 6   land                         int64  
 7   wrong_fragment               int64  
 8   urgent                       int64  
 9   hot                          int64  
 10  num_failed_logins            int64  
 11  logged_in                    int64  
 12  num_compromised              int64  
 13  root_shell                   int64  
 14  su_attempted                 int64  
 15  num_root                     int64  
 16  num_file_creations           int64  
 17  num_shells                   int64  
 18  num_access_files             int64  
 19  num_outbound_cmds            int64  
 20  is_host_login                int64  
 21  is_guest_login               int64  
 22  count                        int64  
 23  srv_count                    int64  
 24  serror_rate                  float64
 25  srv_serror_rate              float64
 26  rerror_rate                  float64
 27  srv_rerror_rate              float64
 28  same_srv_rate                float64
 29  diff_srv_rate                float64
 30  srv_diff_host_rate           float64
 31  dst_host_count               int64  
 32  dst_host_srv_count           int64  
 33  dst_host_same_srv_rate       float64
 34  dst_host_diff_srv_rate       float64
 35  dst_host_same_src_port_rate  float64
 36  dst_host_srv_diff_host_rate  float64
 37  dst_host_serror_rate         float64
 38  dst_host_srv_serror_rate     float64
 39  dst_host_rerror_rate         float64
 40  dst_host_srv_rerror_rate     float64
 41  label                        object 
dtypes: float64(15), int64(23), object(4)
memory usage: 1.5+ GB
```

In [5]:
```
# Checking the distribution of the 'label' column (attack types)
df['label'].value_counts()
```

Out[5]:
```
label
smurf.              2807886
neptune.            1072017
normal.              972781
satan.                15892
ipsweep.              12481
portsweep.            10413
nmap.                  2316
back.                  2203
warezclient.           1020
teardrop.               979
pod.                    264
guess_passwd.            53
buffer_overflow.         30
land.                    21
warezmaster.             20
imap.                    12
rootkit.                 10
loadmodule.               9
ftp_write.                8
multihop.                 7
phf.                      4
perl.                     3
spy.                      2
Name: count, dtype: int64
```

In [6]:
```
# Getting summary statistics for numerical features
df.describe()
```

Out[6]:

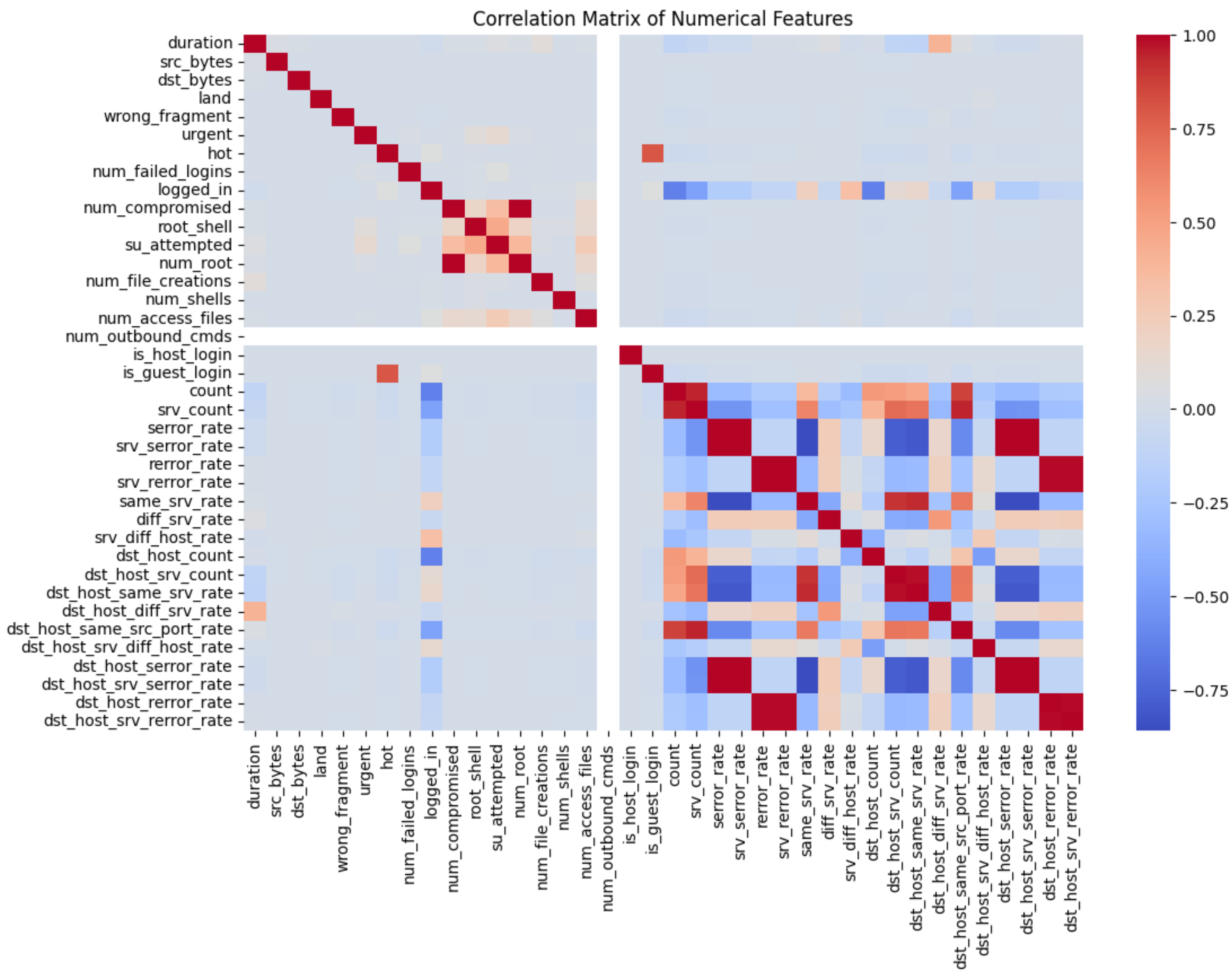
	duration	src_bytes	dst_bytes	land	wrong_fragment
count	4.898431e+06	4.898431e+06	4.898431e+06	4.898431e+06	4.898431e+06
mean	4.834243e+01	1.834621e+03	1.093623e+03	5.716116e-06	6.487792e-04
std	7.233298e+02	9.414311e+05	6.450123e+05	2.390833e-03	4.285434e-02
min	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
25%	0.000000e+00	4.500000e+01	0.000000e+00	0.000000e+00	0.000000e+00
50%	0.000000e+00	5.200000e+02	0.000000e+00	0.000000e+00	0.000000e+00
75%	0.000000e+00	1.032000e+03	0.000000e+00	0.000000e+00	0.000000e+00
max	5.832900e+04	1.379964e+09	1.309937e+09	1.000000e+00	3.000000e+00

8 rows × 38 columns

```
In [7]: # Checking for missing values
missing_values = df.isnull().sum()
print(missing_values[missing_values > 0])
```

```
Series([], dtype: int64)
```

```
In [8]: # Selecting only numeric columns
numeric_columns = df.select_dtypes(include=['number']).columns

# Calculating the correlation matrix only for numeric columns
correlation_matrix = df[numeric_columns].corr()

# Plotting the correlation matrix
plt.figure(figsize=(12, 8))
sns.heatmap(correlation_matrix, cmap='coolwarm', annot=False)
plt.title("Correlation Matrix of Numerical Features")
plt.show()
```



Data Preprocessing

In [9]:
```
# List of DoS attack labels
dos_attacks = ['back.', 'land.', 'neptune.', 'pod.', 'smurf.', 'teardrop.']

# Modifying the 'label' column to classify DoS and Non-DoS
df['label'] = df['label'].apply(lambda x: 'DoS' if x in dos_attacks else 'No

# Displaying the value counts for the new label column
print(df['label'].value_counts())
```

```
label
DoS        3883370
Non-DoS    1015061
Name: count, dtype: int64
```

We defined a list of attack types that are classified as Denial of Service (DoS) attacks. In this case, the DoS attacks are back., land., neptune., pod., smurf., and teardrop. These are specific types of network attacks where the goal is to disrupt the service by overwhelming the system with requests or exploiting vulnerabilities.

In [10]:
```
# One-hot encoding for categorical variables
categorical_columns = ['protocol_type', 'service', 'flag']
df_encoded = pd.get_dummies(df, columns=categorical_columns)

df_encoded.head()
```

Out[10]:

	duration	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot	num_failed_lo
0	0	215	45076	0	0	0	0	
1	0	162	4528	0	0	0	0	
2	0	236	1228	0	0	0	0	
3	0	233	2032	0	0	0	0	
4	0	239	486	0	0	0	0	

5 rows × 123 columns

In [11]:
```
df_encoded = df_encoded.sample(1000, random_state=42)

# Separating features (X) and target (y)
X = df_encoded.drop(columns=['label'])
y = df_encoded['label']

# Applying undersampling to balance the dataset
rus = RandomUnderSampler(random_state=42)
X_resampled, y_resampled = rus.fit_resample(X, y)
```

```
# Displaying the new class distribution after undersampling
print(y_resampled.value_counts())
```

```
label
DoS        199
Non-DoS    199
Name: count, dtype: int64
```

In [12]:
```
# Scaling the resampled data
scaler = MinMaxScaler()
X_resampled_scaled = scaler.fit_transform(X_resampled)
```

In [13]:
```
# Splitting the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_resampled_scaled, y_re
```

Model Training

In [14]:
```
# Kernels to try
kernels = ['linear', 'poly', 'rbf', 'sigmoid']

# Training and evaluating SVM models with different kernels
for kernel in kernels:
    svm = SVC(kernel=kernel, random_state=42)
    svm.fit(X_train, y_train)

    # Making predictions
    y_pred = svm.predict(X_test)

    # Printing the evaluation metrics
    print(f"Kernel: {kernel}")
    print(classification_report(y_test, y_pred))
```

```
Kernel: linear
              precision    recall  f1-score   support

         DoS       0.98      0.98      0.98        46
     Non-DoS       0.97      0.97      0.97        34

    accuracy                           0.97        80
   macro avg       0.97      0.97      0.97        80
weighted avg       0.97      0.97      0.97        80

Kernel: poly
              precision    recall  f1-score   support

         DoS       0.98      0.98      0.98        46
     Non-DoS       0.97      0.97      0.97        34

    accuracy                           0.97        80
   macro avg       0.97      0.97      0.97        80
weighted avg       0.97      0.97      0.97        80

Kernel: rbf
              precision    recall  f1-score   support

         DoS       1.00      0.98      0.99        46
     Non-DoS       0.97      1.00      0.99        34

    accuracy                           0.99        80
   macro avg       0.99      0.99      0.99        80
weighted avg       0.99      0.99      0.99        80

Kernel: sigmoid
              precision    recall  f1-score   support

         DoS       0.92      0.96      0.94        46
     Non-DoS       0.94      0.88      0.91        34

    accuracy                           0.93        80
   macro avg       0.93      0.92      0.92        80
weighted avg       0.93      0.93      0.92        80
```

We train and evaluate SVM models using four different kernels: linear, polynomial (poly), radial basis function (rbf), and sigmoid. The SVC class from scikit-learn is used to create and train the models. Each kernel transforms the data differently, allowing the SVM to capture various types of decision boundaries.

Results and Observations

1. Linear Kernel:

- The accuracy is 0.97, and both classes (DoS and Non-DoS) show high precision, recall, and F1-scores.
- Pros: Computationally efficient, especially for linearly separable data.
- Cons: May not perform well if the data is not linearly separable.

2. Polynomial Kernel:

- Similar performance to the linear kernel with an accuracy of 0.97, indicating that the decision boundary found by the polynomial kernel might not add significant value for this specific dataset.
- Pros: Can capture non-linear patterns if needed.
- Cons: Can be computationally expensive, especially for higher polynomial degrees.

3. RBF (Radial Basis Function) Kernel:

- The best performance with an accuracy of 0.99. The F1-scores for both DoS and Non-DoS classes are 0.99, suggesting that RBF is capturing the underlying patterns in the data more effectively.
- Pros: Great at capturing complex non-linear relationships.
- Cons: Computationally more expensive, and requires tuning parameters like gamma.

4. Sigmoid Kernel:

- The lowest performance with an accuracy of 0.93. Although the precision for both classes is good, the recall for Non-DoS is lower (0.88), indicating that the model misses a higher number of Non-DoS instances compared to other kernels.
- Pros: Sometimes useful for models resembling neural networks.
- Cons: Performance is often unpredictable and can underperform on some datasets.

Conclusion

From the experiments, we observe the following:

- The RBF kernel achieves the best overall performance, with the highest accuracy (0.99) and balanced precision, recall, and F1-scores for both DoS and Non-DoS classes. This suggests that the data is likely non-linearly separable, and the RBF kernel is effective in capturing these complex relationships.
- Both linear and polynomial kernels provide similar performance, achieving high accuracy (0.97). This implies that a linear decision boundary may suffice for this

problem but is not optimal.

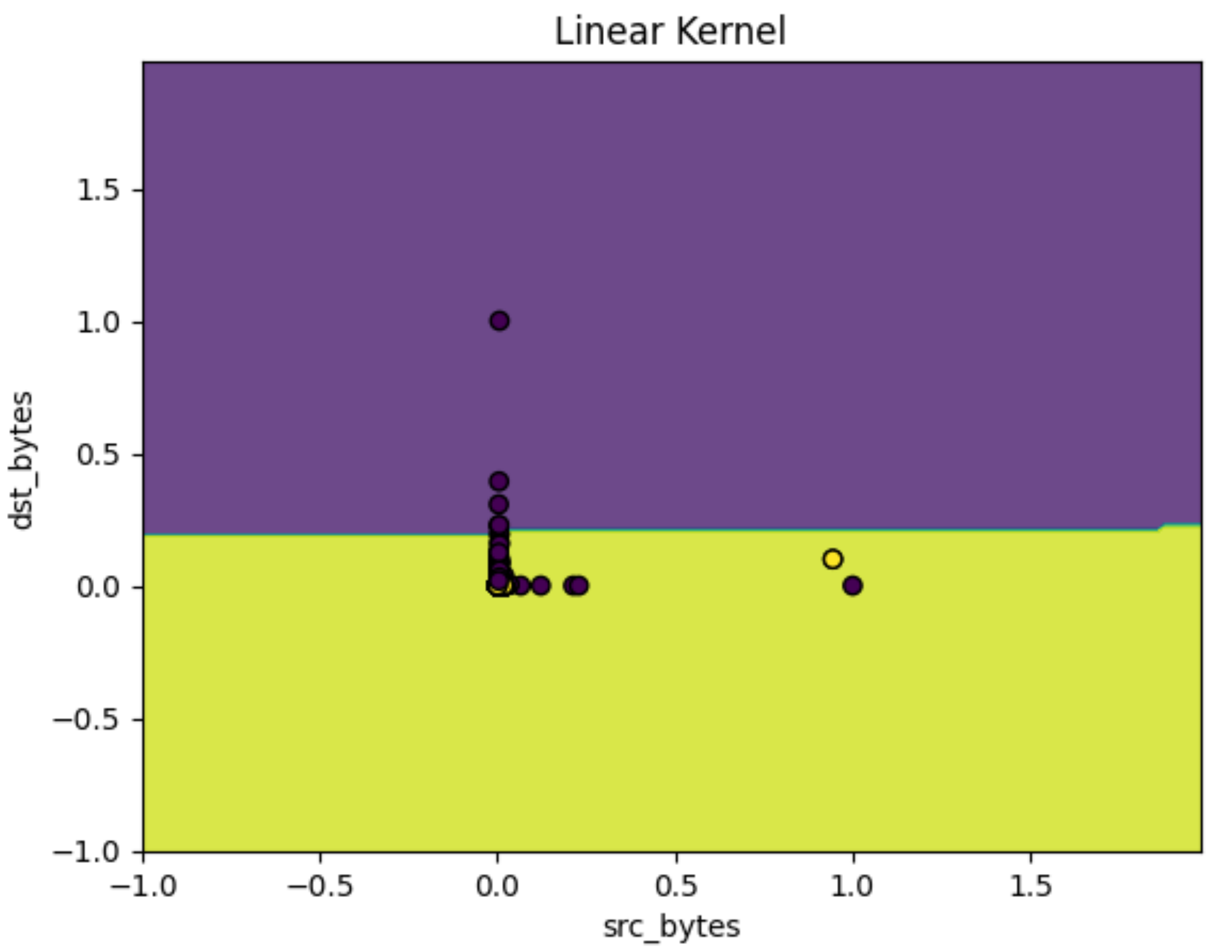
- The sigmoid kernel underperforms compared to the other kernels, likely due to its difficulty in capturing the complex structure of the data.

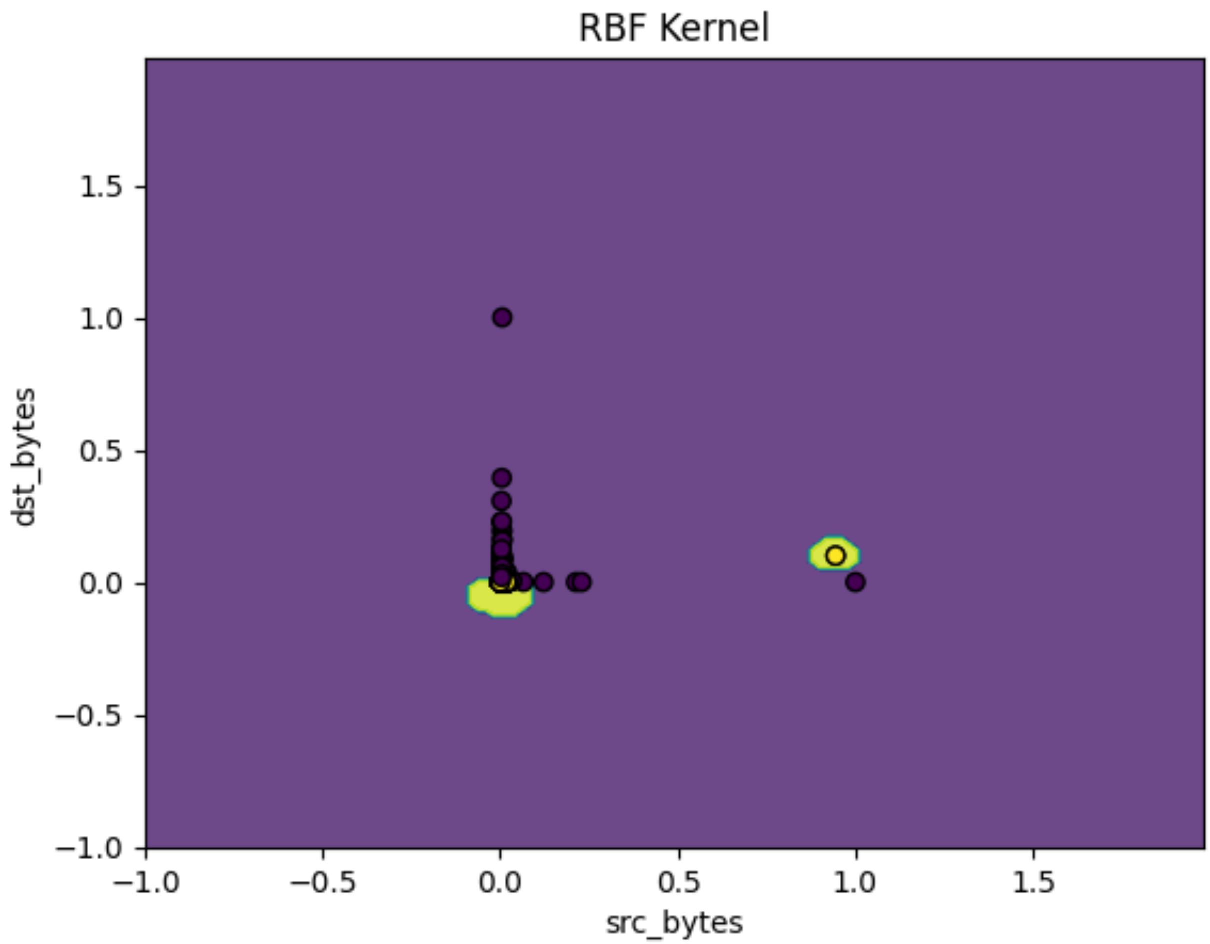
In [15]:
```
# Selecting two features for visualization
df_sample = df_encoded.sample(1000, random_state=42)

X_sample = df_sample[['src_bytes', 'dst_bytes']]
y_sample = df_sample['label']

# Scale the features
X_sample_scaled = scaler.fit_transform(X_sample)
```

The reason for choosing src_bytes and dst_bytes as the two features is because they directly capture the volume of traffic between the source and destination, which is a key characteristic of DoS attacks. DoS attacks typically involve overwhelming a target with a large number of bytes sent from the source (src_bytes), while the destination (dst_bytes) may struggle to respond effectively. These features can help differentiate between normal traffic patterns and the disproportionate traffic volumes seen in DoS attacks. Since DoS attacks often exploit traffic volume to disrupt services, src_bytes and dst_bytes provide crucial insights for distinguishing DoS from Non-DoS activity.

In [16]:
```
# Function to plot decision boundary
def plot_decision_boundary(model, X, y, title):
    h = .02  # Step size in the mesh
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))

    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    plt.contourf(xx, yy, Z, alpha=0.8)
    plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k', marker='o')
    plt.title(title)
    plt.xlabel('src_bytes')
    plt.ylabel('dst_bytes')
    plt.show()

# Converting 'DoS' and 'Non-DoS' labels to 0 and 1
y_sample = np.where(y_sample == 'DoS', 1, 0).astype(int)
X_sample_scaled = np.array(X_sample_scaled)
y_sample = np.array(y_sample)

# Training the SVM models with Linear and RBF kernels
```

```
svm_linear = SVC(kernel='linear', random_state=42)
svm_rbf = SVC(kernel='rbf', random_state=42)

# Fitting the models
svm_linear.fit(X_sample_scaled, y_sample)
svm_rbf.fit(X_sample_scaled, y_sample)

# Plotting for linear kernel
plot_decision_boundary(svm_linear, X_sample_scaled, y_sample, "Linear Kernel

# Plotting for RBF kernel
plot_decision_boundary(svm_rbf, X_sample_scaled, y_sample, "RBF Kernel")
```





Linear Kernel Decision Boundary The first plot shows the decision boundary for the linear kernel. The boundary is a straight line, separating the feature space into two regions: one for DoS attacks (yellow region) and the other for Non-DoS attacks (purple region). However, from the visualization, it is clear that the linear kernel struggles to separate some points cleanly. The misclassified points can be observed as data points that fall into the “wrong” regions.

The linear decision boundary is relatively simple and divides the space into two halves. However, some overlap between classes suggests that a linear kernel may not be sufficient to capture the complexity of the relationship between src_bytes and dst_bytes for differentiating DoS and Non-DoS attacks.

The linear kernel might not be ideal for this classification problem, as it assumes the data is linearly separable, which does not appear to be the case for these two features.

RBF Kernel Decision Boundary The second plot shows the decision boundary for the RBF kernel. Unlike the linear kernel, the RBF kernel produces a more flexible, non-linear boundary that better adapts to the clusters in the feature space. The boundary surrounds clusters of DoS or Non-DoS data points, capturing their non-linear relationships more effectively.

The decision boundary for the RBF kernel is more complex and better fits the data, particularly in areas where the linear model struggled. It encloses the clusters of points more tightly, reducing the number of misclassifications.

The RBF kernel is clearly more suited for this dataset, as it captures the non-linear patterns between the src_bytes and dst_bytes features, resulting in better class separation.

In [17]:

```
X = df_encoded[['src_bytes', 'dst_bytes']].values
y = df_encoded['label'].apply(lambda x: 1 if x == 'DoS' else 0).values

# Scale the features for better performance in SVM
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)

# Splitting the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0

# SVM regularization parameter
C = 1.0
```

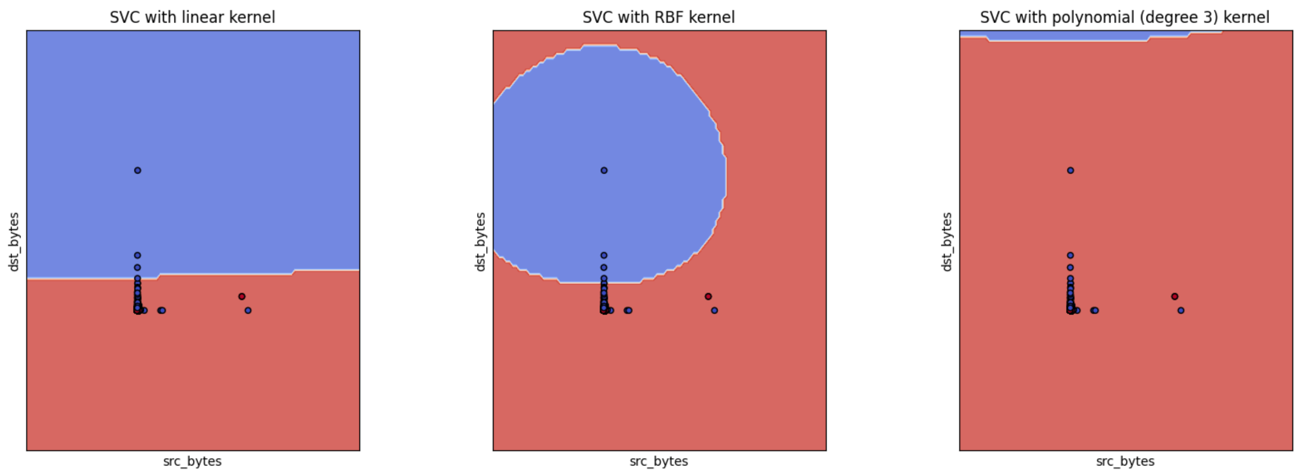
In [18]:

```
# Defining SVM models with different kernels
models = (
    SVC(kernel="linear", C=C),
    SVC(kernel="rbf", gamma=0.7, C=C),
    SVC(kernel="poly", degree=3, gamma="auto", C=C),
)

# Fitting the models on the training data
models = [clf.fit(X_train, y_train) for clf in models]
```

In [19]:

```
titles = (
    "SVC with linear kernel",
    "SVC with RBF kernel",
    "SVC with polynomial (degree 3) kernel",
)

fig, sub = plt.subplots(1, 3, figsize=(18, 6))
plt.subplots_adjust(wspace=0.4, hspace=0.4)

X0, X1 = X_train[:, 0], X_train[:, 1]

# Plotting decision boundary for each SVM model
for clf, title, ax in zip(models, titles, sub.flatten()):
```

```
    disp = DecisionBoundaryDisplay.from_estimator(
        clf,
        X_train,
        response_method="predict",
        cmap=plt.cm.coolwarm,
        alpha=0.8,
        ax=ax,
        xlabel="src_bytes",
        ylabel="dst_bytes",
    )
    ax.scatter(X0, X1, c=y_train, cmap=plt.cm.coolwarm, s=20, edgecolors="k"
    ax.set_xticks(())
    ax.set_yticks(())
    ax.set_title(title)

plt.show()
```



Linear Kernel:

The decision boundary created by the linear kernel is a straight line that separates the feature space into two regions (DoS and Non-DoS). While this works for simple, linearly separable data, it can be seen from the plot that some data points overlap across the boundary, leading to potential misclassification.

The linear kernel is unable to capture more complex, non-linear patterns in the data, and thus provides a relatively simple decision boundary.

The linear kernel may not be the best choice when the data is not linearly separable, as is the case here.

RBF Kernel:

The decision boundary for the RBF kernel is circular and much more complex than the linear kernel. This kernel captures the non-linear relationships between the features by drawing boundaries that tightly enclose groups of data points, particularly those from the DoS class.

The RBF kernel is clearly more flexible and performs better in separating the data points, especially in regions where the classes are densely packed.

The RBF kernel is well-suited for this dataset, where non-linear decision boundaries are necessary to distinguish between DoS and Non-DoS classes.

Polynomial Kernel (degree 3):

The polynomial kernel of degree 3 creates a more complex boundary compared to the linear kernel but is still relatively simple compared to the RBF kernel. The decision boundary shows slight curvature, but it does not fully capture the complexity of the data in the same way that the RBF kernel does.

The polynomial kernel provides some improvement over the linear kernel but is not as flexible as the RBF kernel. There are still data points that fall on the wrong side of the decision boundary.

Conclusion: The polynomial kernel may offer a compromise between the linear and RBF kernels, but it is still less effective at handling highly non-linear relationships.

Final Conclusion

The decision boundaries generated by these three kernels illustrate how the choice of kernel affects the ability of the SVM model to classify the data. The linear kernel provides a simple, computationally efficient solution but lacks the flexibility needed for more complex data. The RBF kernel demonstrates superior performance, capturing the non-linear patterns in the data and creating more accurate class boundaries. The polynomial kernel lies somewhere in between, offering some flexibility but not to the extent of the RBF kernel.

For this particular dataset and feature selection (src_bytes and dst_bytes), the RBF kernel emerges as the best choice, creating the most accurate decision boundary for distinguishing between DoS and Non-DoS attacks.