Ahmed Awadallah

Support Vector Machines and Stochastic Gradient Descent

Introduction

Security has always been a concern for everyone at all times. In the modern age the problem is security at scale. Specifically with internet access the number of people that want to connect to a certain domain is impossible to do manually at a certain point. In order to continue scaling or to handle a scaling number of attacks automated tools must be created.

The dataset for this case study contains 65532 observations with a total of 11 features and 1 target.

Methods

The dataset is a single file so it is simply loaded into the program.

The dataset was shuffled to remove any impact of initial ordering in the dataset,

N/A values did not exist within the dataset so no imputation was performed.

Inspection of the data reveals that all the features are numeric in nature. Some of these “numeric” features are better represented as a categorical feature in reality so these features were one hot encoded. The features that were transformed by this process were “Source Port”, “Destination Port”, “NAT Source Port”, and “NAT Destination Port”. The rest remained as numeric as that representation made the most sense. This led to the dataset now having 57688 features.

Numeric variables were scaled as the models used are sensitive to data scale.

The stochastic gradient descent model (SGD) was tuned using randomized search cross validation with 5 folds. The optimizing metric is accuracy as this was stated in class. The search space is listed below.

```
distributions = {
    "alpha": np.logspace(-6, 6, 16),
    "l1_ratio": np.logspace(-6, 0, 16)
}
```

Figure 1: SGD Search Space

The final model's hyperparameters are below.

```
SGD_params = {
    "loss": "log_loss",
    "penalty": "elasticnet",
    "early_stopping": True,
    "n_iter_no_change": 5,
    'l1_ratio': 1.0,
    'alpha': 6.30957344480193e-06
}
```

Figure 2: Final SGD Hyperparameters

The support vector machine model (SVM) was tuned using randomized search cross validation with 5 folds. The optimizing metric is also accuracy as it was stated in class. The search space is listed below.

```
distributions = {
    "C": np.logspace(-6, 6, 16),
    "gamma": ["scale", "auto"],
    "kernel": ["rbf", "poly"],
    "degree": [3, 4, 5, 6]
}
```

Figure 3: SVM Search Space

The final hyperparameters are below.

```
SVC_params = {'kernel': 'poly',
              'gamma': 'scale',
              'degree': 3,
              'C': 1000000.0}
```

Figure 4: SVC Final Hyperparameters

The SGD final model was fit using the entire dataset but the SVM model was fit using a subset of the data. This choice is because the SVM model takes too long to compute on the dataset. The time complexity of the SVM is approximated to be between n^2 and n^3. We assumed the worst case of n^3 and solved the equation t=An^3 with a trial run. The final equation resulted in t=(5.13e^-8)n^3. Using this equation we aimed for a run that would take around 14 hours which resulted in a sample size of around 10000. A proper 5 fold cv split would result in a computation time of 85 days according to this equation

Results

SGD Classification Report

SGD Classification Report

	precision	recall	f1-score	support
allow	1.00	1.00	1.00	7528
deny	1.00	0.99	0.99	3001
drop	1.00	1.00	1.00	2564
reset-both	1.00	0.79	0.88	14
accuracy			1.00	13107
macro avg	1.00	0.94	0.97	13107
weighted avg	1.00	1.00	1.00	13107

Figure 5: SGD Classification Report

The results of the SGD model are overall near perfect. The three popular classes have essentially perfect predictions. Reset-both had non perfect results but this is likely due to how rare it is(look at the support count). Despite this the reset-both class was predicted decently well though take this with a grain of salt as the number of observations is miniscule. From the precision and recall scores for the popular classes the model shows it's not just accurate but also has no tradeoff between either which further shows the strength of the model.

SGD Feature Importance

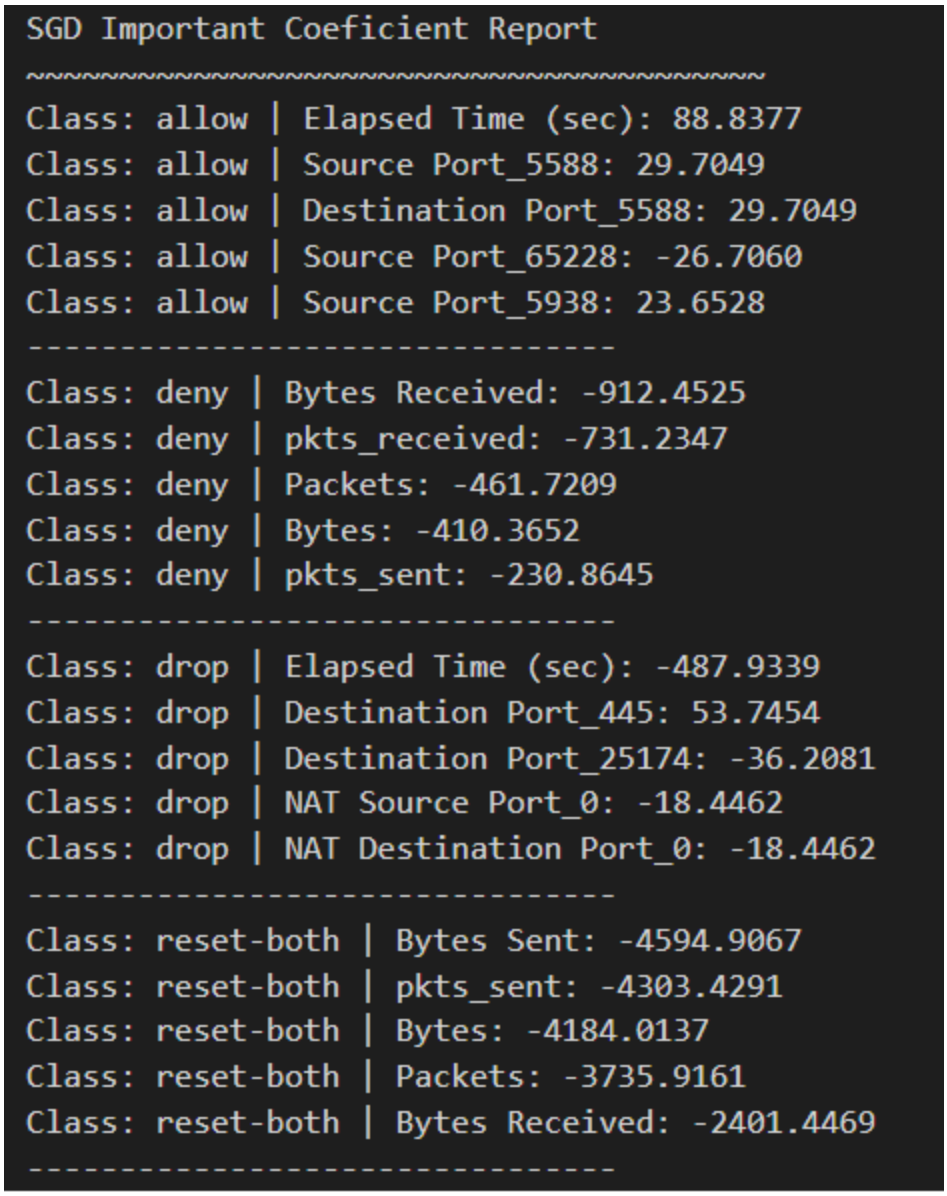

Figure 6: SGD Feature Importance Report

Above is the feature importances per class for the SGD model.

The allow class has a moderate positive relationship with elapsed time and some weak relationships with various source and destination ports. The weak relationships with the source ports make sense as technically these are chosen at random and shouldn't give any information. Overall interpretation is that as elapsed time increases the odds that the request falls under the allowed class.

The deny class has a large negative relationship with all its major features. They mostly have to do with things related to the amount of data being received. Overall sentiment can be summed

up as the increasing amount of data being received reduces the odds a connection is denied. Conversely, connections that receive a small amount of data are higher odds of being dropped.

The drop class has a strong negative relationship with elapsed time and various weak relationships with destination ports and NAT source port. Overall interpretation is that short connections have higher odds for being dropped. Additionally, since “Destination Port_445” is decently large, an interpretation of this is that if the destination port is the exact number 445 there is an increase in the odds that the connection is dropped. I’m not 100% sure this is related but windows machines have a destination port 445 and these ports are commonly associated with malware attacks. They are normally blocked at the firewall level. This positive association seen in the model could be a reflection that this port commonly drops connections for this reason. (https://www.speedguide.net/port.php?port=445 for info)

Lastly the “reset-both” class has large negative relationships with the amount of data sent. This can be interpreted as the more data is being sent results in the odds of a reset-both event. This class has immense relationships to these features compared to the other classes.

SGD Confusion Matrix Heatmap

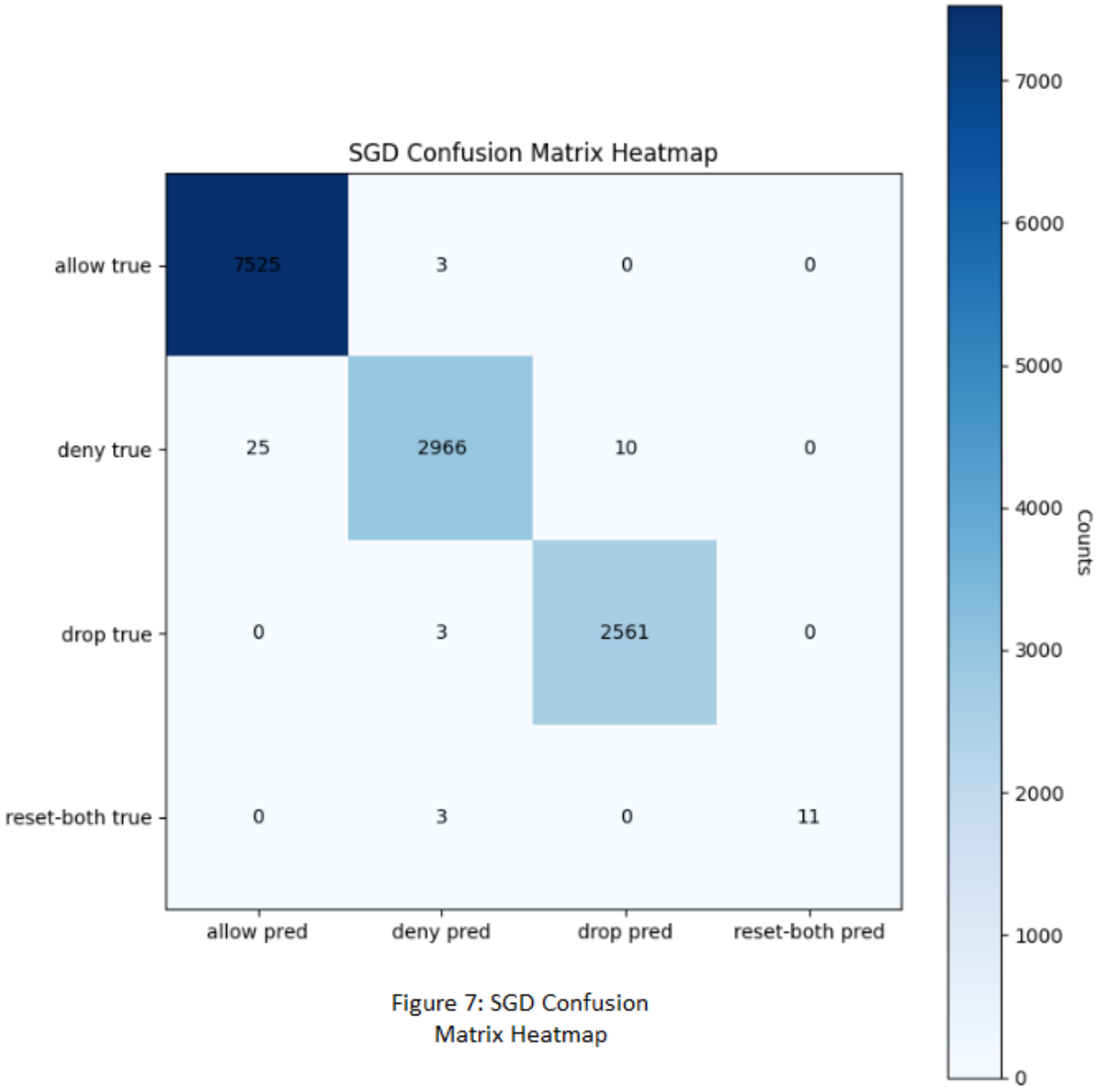

Figure 7: SGD Confusion Matrix Heatmap

Above is the SGD Confusion Matrix represented as a heatmap. Overall we can see that near perfect performance as all the values are within the diagonal. There are a few misclassified results but they are too small to really derive meaning from. One could perhaps say that denying and allowing a connection have a small overlap as well as denying and dropping.

SVM Classification Report

SVM Classification Report

	precision	recall	f1-score	support
allow	1.00	1.00	1.00	31971
deny	0.99	1.00	0.99	12745
drop	1.00	1.00	1.00	10937
reset-both	0.00	0.00	0.00	50
accuracy			1.00	55703
macro avg	0.75	0.75	0.75	55703
weighted avg	1.00	1.00	1.00	55703

Figure 8: SVM Classification Report

Overall the SVM model does well in the three popular classes but completely fails in the rare reset-both class. The reset-both class is likely enveloped completely by one of the popular classes.

Precision and recall reveal that there is no tradeoff between these two metrics for the popular classes.

Additionally remember that since we only trained on 10000 samples we used the rest of the dataset for evaluation. This can be seen with the large support values for each class. While this limited the model it provides a decent confidence that there are unique and easily separable patterns in the popular classes.

SVM Feature Importance

Nonlinear SVMs are not interpretable so a feature importance for this model has not been provided.

SVM Confusion Matrix Heatmap

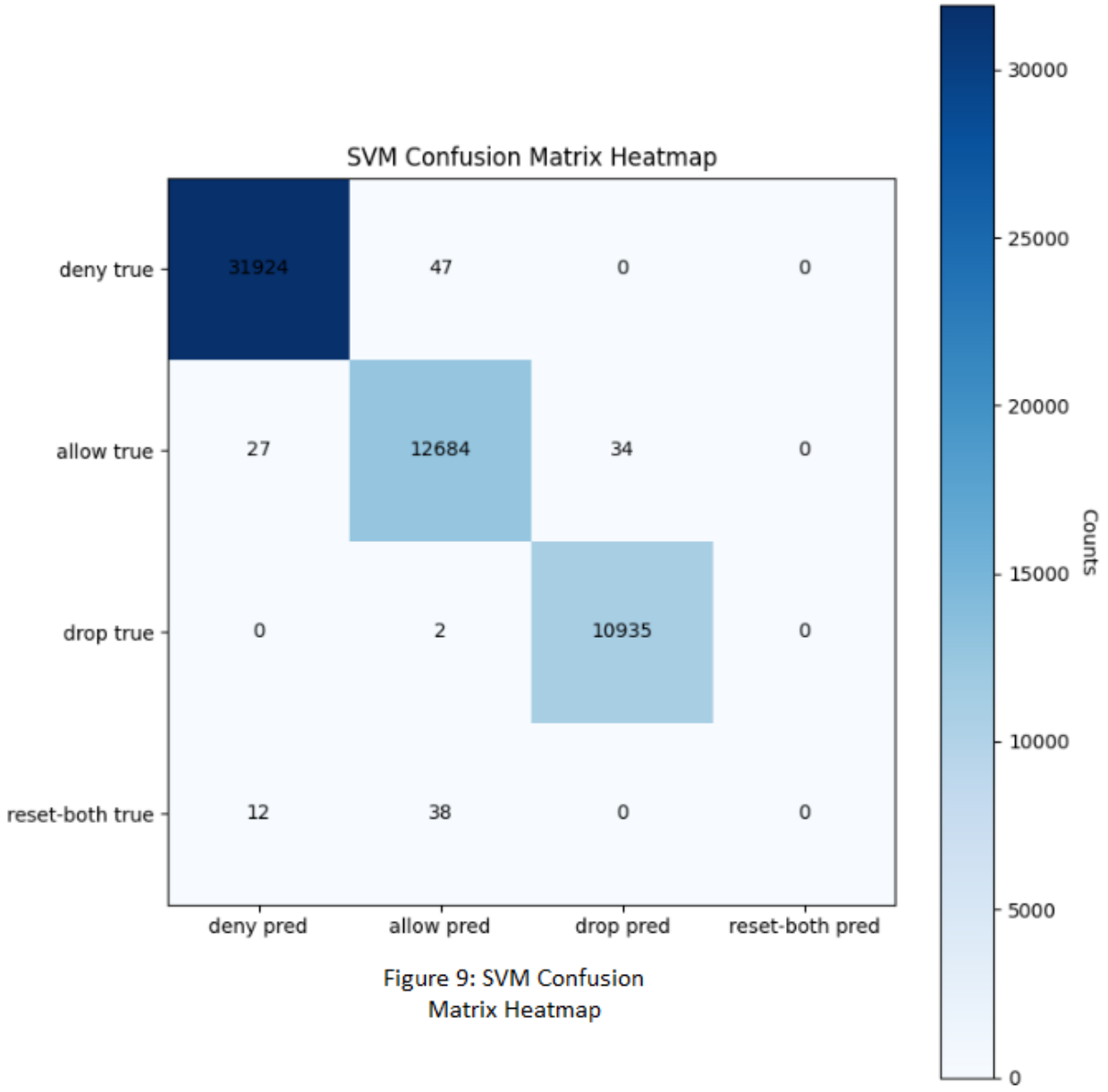

Figure 9: SVM Confusion Matrix Heatmap

Above is the SVM confusion matrix heat map. Overall for the popular class it does well. This plot also confirms our suspicions that the reset-both class is enveloped by other classes. In this case we see that the reset-both class is somewhere between the deny and allow space.

SVM SGD Comparison Commentary

Overall the SGD model performed better than SGD. While SGD and SVM have high accuracy scores the SGD provides the additional benefit of being able to provide predictions for the rare “reset-both” class.

Misc Commentary

The models both performed well but I wanted to call into question the validity of the model in practice. Multiple features would only be known after a connection has been completed rather than at the start.

For example the Elapsed Time (sec) feature is only known when the connection is completely finished. The expectation of a model is to decide whether or not to allow or drop a connection but the model can't possibly know how long the connection is going to take at the start of the connection. This means this feature gives information that would not be available in practice. Features that are similar to Elapsed Time are pkts_received, pkts_sent, packets, Bytes Received, Bytes Sent, and Bytes. Removal of these features would leave only source and destination ports which is not enough to make accurate predictions as evidenced by the feature importances. All is not lost though, what is clear is that the end state of a connection results in attributes that are clearly separable.

In practice I believe a model that could achieve automated security would be some kind of Bayesian model that evaluates the connection over time. The model would be able to observe how the connection evolves with time then make a judgment call on whether to drop, continue, reset, or deny.

Conclusion

It's clear that SVMs are powerful tools but their weakness at scaling is an immense hindrance to its applicability. Not only is the time to compute too much but it also prevents the model from performing in datasets with rare targets. For this report it's possible that SVM could have performed better predicting "reset-both" but because training can only happen on a subset of the data it does not have enough to predict. With that said this is a limitation of exact SVM computation. For this report other methods were not explored but the experience serves as a reason to use them instead of sklearn.

SGD on the other hand showed that where SVM fails at scaling it excels while also still outputting strong results. The only concern for this model is memory but even this can be overcome through more modern tools.

Code

Appendix A: Script

```
import sys
import os
import pandas as pd
import numpy as np
from sklearn.metrics import classification_report
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.linear_model import SGDClassifier


class UnexpectedFileStructureError(Exception):
    def __init__(self):
        msg = "This script expects a file path to a directory that contains "
        msg += "CS5 Network Dataset. The Network dataset is a "
        msg += "single csv labelled as log2.csv "
        msg += "Did you specify a folder path that "
        msg += "contains this file?"
        super().__init__(msg)


class MissingInputError(Exception):
    def __init__(self):
        msg = "This script expects atleast one input, a file directory that "
        msg += "contains the network dataset. You input nothing. "
        msg += "Did you forget to add an argument with the script?"
        super().__init__(msg)
```

```python
class ExcessiveInputError(Exception):
    def __init__(self):
        msg = "This script expects atleast one input and at most 2."
        msg += " The first argument is the file directory containing the data"
        msg += " and the second optional one is a flag specifying if you want"
        msg += " to train on all the data or a subset"
        msg += " You called this script with more than two arguments."
        msg += " Only use one or two and try again."
        super().__init__(msg)


class UnexpectedFlagError(Exception):
    def __init__(self):
        msg = "Flag input can only have a value of 0 or 1. Script received"
        msg += " something unexpected instead."
        super().__init__(msg)


class NonexistantFolderError(Exception):
    def __init__(self):
        msg = "Input folder does not exist"
        super().__init__(msg)
```

```
def check_correct_file_structure(inputs):
    data_direc = inputs[1]
    expected_files = ["log2.csv"]
    if os.path.exists(data_direc):
        for root, dirs, files in os.walk(data_direc):
            if files == []:
                raise UnexpectedFileStructureError
            for f in files:
                if f not in expected_files:
                    print(f)
                    raise UnexpectedFileStructureError
    else:
        raise NonexistantFolderError


def check_correct_flag_input_val(inputs):
    flag = inputs[2]
    if (flag != "1") and (flag != "0"):
        raise UnexpectedFlagError


def check_valid_inputs(inputs):
    if len(inputs) < 2:
        raise MissingInputError
    elif len(inputs) > 3:
        raise ExcessiveInputError
```

```
def load_data(inputs):
    data_direc = inputs[1]
    for root, dirs, files in os.walk(data_direc):
        for f in files:
            file_path = os.path.join(root, f)
            raw_data = pd.read_csv(file_path)
    return raw_data


def get_flag(inputs):
    flag = inputs[2]
    if flag == "1":
        flag = 1
    elif flag == "0":
        flag = 0
    return flag
```

```python
def plot_heatmap(data,
                 xlab,
                 ylab,
                 size=9,
                 title="untitled heatmap",
                 cbar_label="untitled",
                 save_name="untitled.png"):

    fig, ax = plt.subplots(figsize=(size, size))
    im = ax.imshow(data, cmap=plt.cm.Blues)

    for i in range(len(ylab)):
        for j in range(len(xlab)):
            ax.text(j, i, data[i, j],
                    ha="center", va="center", color="black")

    # Show all ticks and label them with the respective list entries
    ax.set_xticks(np.arange(len(xlab)), labels=xlab)
    ax.set_yticks(np.arange(len(ylab)), labels=ylab)

    cbar = ax.figure.colorbar(im, ax=ax)
    cbar.ax.set_ylabel(cbar_label, rotation=-90, va="bottom")

    ax.set_title(title)
    fig.tight_layout()
    plt.savefig(save_name, bbox_inches='tight')
    plt.clf()
```

```python
def get_top_n_coef_multiclass(feature_list, coef_list, n=5):
    feature_scores_abs = {}
    top_n_per_class = []
    feature_scores = {}
    class_count = coef_list.shape[0]
    for class_id in range(class_count):
        for i in range(len(feature_list)):
            feature = feature_list[i]
            coef = coef_list[class_id, i]
            feature_scores_abs[feature] = abs(coef)
            feature_scores[feature] = coef

        feature_scores_abs = dict(sorted(feature_scores_abs.items(),
                                         key=lambda item: item[1], reverse=True))
        top_n_features = list(feature_scores_abs.keys())[0:n]

        top_feature_scores = {}
        for feature in top_n_features:
            top_feature_scores[feature] = feature_scores[feature]

        top_n_per_class.append(top_feature_scores)

    return top_n_per_class
```

```python
def top_coefs_multiclass_report(top_n_coefs, id_targ_map):
    str_report = ""
    for class_id in range(len(top_n_coefs)):
        feature_top_n = top_n_coefs[class_id]
        class_name = id_targ_map[class_id]
        for feature in feature_top_n.keys():
            coef = feature_top_n[feature]
            str_report += f"Class: {class_name} | {feature}: {coef:.4f}\n"
        str_report += "------------------------------\n"
    return str_report


def log_section(log_file, title="No Title", content="No Content"):
    log_file.write(title + "\n")
    log_file.write("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n")
    log_file.write(content + "\n")
    log_file.write("\n")


if __name__ == '__main__':
    inputs = sys.argv
    check_valid_inputs(inputs)
    check_correct_file_structure(inputs)
    full_run_flag = 0
    if len(inputs) == 3:
        check_correct_flag_input_val(inputs)
        full_run_flag = get_flag(inputs)
    else:
        info_message = "This script by default runs on a small sample size"
        info_message += " by default to save"
        info_message += " computation time and memory.\n"
        info_message += " A full run can be computed by inputting 1 as"
        info_message += " the second script argument"
        print(info_message)
```

```
if full_run_flag:
    print("Performing a full run...")
else:
    print("Performing an example run...")

# Mappings
action_to_id = {"allow": 0, "deny": 1, "drop": 2, "reset-both": 3}
id_to_action = {0: "allow", 1: "deny", 2: "drop", 3: "reset-both"}

# Log Start
log_file = open("log.txt", "w")
log_file.write("Case Study 5 Report\n\n")

# Use input to load data
raw_data = load_data(inputs)

# Shuffle Data and sample
if full_run_flag:
    raw_data = raw_data.sample(frac=1)
else:
    raw_data = raw_data.sample(frac=0.07)
```

```
# Preprocessing
clean_df = raw_data.copy()
categ_features = ["Source Port",
                  "Destination Port",
                  "NAT Source Port",
                  "NAT Destination Port"]
target_feature = ["Action"]
cts_features = ["Bytes",
                "Bytes Sent",
                "Bytes Received",
                "Packets",
                "Elapsed Time (sec)",
                "pkts_sent",
                "pkts_received"]
scaler = StandardScaler()
clean_df[cts_features] = scaler.fit_transform(clean_df[cts_features])
y = clean_df[target_feature[0]]
x = clean_df.drop(target_feature, axis=1)
for categ_feature in categ_features:
    x[categ_feature] = x[categ_feature].astype("category")
x = pd.get_dummies(x)

# Split emulating a 5 fold cv
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2)
```

```python
# SGD
SGD_params = {
    "loss": "log_loss",
    "penalty": "elasticnet",
    "early_stopping": True,
    "n_iter_no_change": 5,
    'l1_ratio': 1.0,
    'alpha': 6.30957344480193e-06
}
sgd_clf = SGDClassifier(**SGD_params)
print("SGD TRAIN START")
sgd_clf.fit(x_train, y_train)
print("SGD TRAIN END")

# Classif Report SGD
y_pred_final = sgd_clf.predict(x_test)
classif_rep = classification_report(
    y_test,
    y_pred_final
)
log_section(log_file,
            title="SGD Classification Report",
            content=classif_rep)
```

```python
# Confusion Matrix SGD
conf_mat = confusion_matrix(y_test, y_pred_final)

base_labels = y_test.unique()
label_true = []
label_pred = []
for i in range(len(base_labels)):
    true_label = base_labels[i] + " true"
    pred_label = base_labels[i] + " pred"
    label_pred.append(pred_label)
    label_true.append(true_label)
plot_heatmap(conf_mat,
             label_pred,
             label_true,
             size=8,
             title="SGD Confusion Matrix Heatmap",
             cbar_label="Counts",
             save_name="SGD_conf_heatmap.png")

# Feature Importance SGD
class_top_n = get_top_n_coef_multiclass(
    list(x.columns),
    sgd_clf.coef_,
    n=5)
coefs_rep = top_coefs_multiclass_report(
    class_top_n,
    id_to_action
)
log_section(log_file,
            title="SGD Important Coeficient Report",
            content=coefs_rep)
```

```python
# SVM
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.85)
SVC_params = {'kernel': 'poly',
              'gamma': 'scale',
              'degree': 3,
              'C': 1000000.0}
svm_clf = SVC(**SVC_params)
print("SVM TRAIN START")
svm_clf.fit(x_train, y_train)
print("SVM TRAIN END")

# Classif Report SVM
y_pred_final = svm_clf.predict(x_test)
classif_rep = classification_report(
    y_test,
    y_pred_final
)
log_section(log_file,
            title="SVM Classification Report",
            content=classif_rep)
```

```python
# Confusion Matrix SVM
conf_mat = confusion_matrix(y_test, y_pred_final)

base_labels = y_test.unique()
label_true = []
label_pred = []
for i in range(len(base_labels)):
    true_label = base_labels[i] + " true"
    pred_label = base_labels[i] + " pred"
    label_pred.append(pred_label)
    label_true.append(true_label)
plot_heatmap(conf_mat,
             label_pred,
             label_true,
             size=8,
             title="SVM Confusion Matrix Heatmap",
             cbar_label="Counts",
             save_name="svm_conf_heatmap.png")

# SVM Feature Importance not here cuz nonlinear model

# Fin
log_file.write("\n")
log_file.close()
```