Ryan Herrin, Luke Stodgel
DS7333
03/09/2023

Internet Connection Request Prediction using Support Vector Machines and Stochastic Gradient Descent

Introduction

Predicting internet connection requests can be useful in a variety of applications. Two example applications are in network planning and fraud detection. With network planning, being able to predict future internet connection requests can help network planners and engineers to design and optimize network infrastructure to meet demand. As for fraud detection, by being able to detect or predict suspicious internet activity, banks can flag that activity to protect their customers. Overall, predicting internet connection requests can help organizations improve network performance, optimize resource allocation, provide better customer service, and improve security.

In this project we used Support Vector Machine (SVM) and Stochastic Gradient Descent (SGD) models to predict the class of an internet connection request. This was a multi-class classification task.

Methods

Our dataset contained features pertaining to internet traffic that was passed through a firewall. The shape of the dataset was 65,532 rows by 12 columns. Our target variable, named “action”, contained four possible values, “allow”, “deny”, “drop”, and “reset-both”.

Here is a list of a few of the columns and their meanings (please find the full list of features in the appendix):

- Source Port: The port number on the sender's machine that is used to send the traffic.
- Destination Port: The port number on the receiver's machine that is used to receive the traffic.
- Bytes: The total number of bytes in the network traffic.
- Packets: The total number of packets in the network traffic.
- Elapsed Time (sec): The time duration for which the network traffic was observed.

Data Preprocessing

To start, we read in our dataset (log2.csv) using python’s built-in csv library. From there we saved the data in a dictionary where the independent data and target variable data were stored in two separate numpy arrays.

We chose to remove (drop) the Source Port, and NAT source port features. We did this because source port numbers are randomly generated when connections are established, act as session ID’s, and thus are objectively meaningless in this dataset (see https://en.wikipedia.org/wiki/Ephemeral_port for more information). If we did not remove the source port columns and one hot encoded them, the source port’s randomly generated numbers would not have only added thousands of columns to our dataset,

adding noise and interfering with the results, but also would have exponentially increased the time it would have taken to fit models. We feel confident with this decision because we tested this theory by creating models that included the one-hot encoded source ports, and for our SVM model our accuracy dropped from 96.7% to between 85-90% and for our SGD model the accuracy dropped from 96.7% to between 75-80%.

Next, we one-hot encoded the Destination Port and the NAT Destination Port columns using Sklearn's OneHotEncoder package. Before one-hot encoding these two categorical columns we compressed the data size by including only the ports that made up 97% of the Destination and NAT destination ports. The remaining 3% from each of these columns separately was put into their own "others" column. This resulted in a reduction of Destination Port values from 3,336 down to 587. For NAT destination ports the number of ports went from 2,540 down to 95. In total, one-hot encoding those two columns resulted in an additional 678 columns to our training set.

The continuous data (Bytes, Bytes Sent, Bytes Received, Packets, Elapsed Time (sec), pkts_sent, pkts_received) were all scaled using the Sklearn StandardScaler package. Scaling is important because in models like SVM (SVC in Sklearn) and SGD, the models find a decision boundary between two points and these models have an easier time measuring that distance when the data is scaled as opposed to when it is not.

The default data type of our continuous data was float64, and in order to reduce the amount of memory used during model fitting, we converted it to float32. This also saved us time during model fitting. We used the float data type because scaling our data introduced decimal values.

Our final dataset contained 65,532 rows with 687 feature columns and the target (action) column.

Addressing the Missing Data

There was no missing data in this dataset.

Data Sampling and GridsearchCV

Data sampling allowed us to use GridSearchCV more efficiently in this project. We used one random sample of 10,000 rows in our SVM gridsearch and then another separate random sample of 10,000 rows in our SGD gridsearch.

We used gridsearchcv only to find the best regularization terms for our SVM and SGD models. Once we had those, we manually tested a few different other parameters before declaring our models final. For our final SVM model we specifically tested linear, poly and rbf kernels to see if there was an impact on performance. We will comment on the impact of trying other kernels later in this paper. For our final SGD model, we manually tested hinge, modified_huber loss functions, and l2 and l1 penalties. We comment on whether or not these parameters impacted our models' performance later in this paper.

GridSearchCV for SVM

Our SVM gridsearchcv parameter grid looked like this:

```
param_grid = {
    'C': [.000001, .00001, .0001, .001, .01, .1, 1.0, 5, 10, 20]
}
```

We tested 10 different values of C. C is the regularization term in SVM. A higher value of C results in weaker regularization in SVMs.

Within our gridsearch algorithm we also used the parameters: scoring='accuracy', n_iter=5, cv=5 and n_jobs=-1. We picked accuracy as our scoring metric because obtaining the highest accuracy was the objective of this project.

The best parameter returned by our SVM gridsearchcv was: 'C': 20. And, using this C value and the rest of the model parameters set to their default values, we received a 0.997 accuracy score. In our final model we ended up using C=1 instead of C=20 because we were still receiving 0.997 accuracy but by using a lower C value, we reduced the risk of our model being overfit.

Final SVM Model Parameters

The parameters we used for our final SVM model looked like this:

```
best_params_svm = {
      'C': 1, 'kernel': 'rbf',
      'class_weight': None, 'decision_function_shape': 'ovr'
}
```

We realize that all of these values are the default values for the SVC() algorithm. We tried several other values of C, many different kernels (mentioned in section "Data Sampling and GridsearchCV"), setting class_weight to balanced, and even changing decision_function_shape to ovo. None of that improved the performance of our model. We did not fiddle with the parameters "degree" or "gamma" because we were advised that the default values for these would work fine; And they did, as shown in the results section.

We picked the rbf kernel over the linear kernel (which performed just as well as rbf) because in general, rbf can model non-linear decision boundaries that are not possible with a linear kernel. Also, rbf usually performs better with higher-dimensional data compared to the linear kernel.

GridSearchCV for SGD

For our SGD GridSearchCV, our parameter grid looked like this:

```
param_grid = {
    'alpha': [1e-10,1e-9,1e-8,1e-7,1-6,1e-5,1e-4,1e-3,1e-2,1e-1,1]
}
```

We also used parameters n_iter=5, scoring='accuracy', cv=5, n_jobs=-1. We picked accuracy as our scoring metric because, similar to before, obtaining the highest accuracy was the objective of this project.

The best parameter returned by our SGD gridsearchcv was: 'alpha': 1e-6. And, using this alpha value and the rest of the model parameters set to their default values, we received a 0.9968 accuracy score.

Final SGDModel Parameters

The parameters we used for our final SGD model looked like this:

```
best_params_sgd = {
      'loss': 'modified_huber', 'penalty': 'l2', 'alpha': 1e-6,
      'max_iter': 1000, "learning_rate": 'optimal',
      'early_stopping': True, 'n_iter_no_change': 10,
```

```
        'class_weight': None
    }
```

Similarly to our final SVM model parameters, we used mostly default values because, with them, we were able to make an extremely accurate model. We didn't use all default parameters though. We used the modified_huber instead of the hinge loss function. We did this because modified_huber increased the performance of our model consistently by .0001 accuracy, precision, recall and f1_score. We also enabled early_stopping to prevent overfitting and to hopefully improve the model's generalization performance on unseen data. We tested out a few values for the n_iter_no_change variable and found that 10 gave us the best, most consistent performance. Also, similar to our SVM model, setting class_weight to "balanced" consistently decreased the performance of the model. So it was left as the default, "None".

Prediction With or Without using Predicted Probabilities?

In this project we opted to only make predictions using the traditional multi-class prediction method within the predict() function. We believe our final models performed well enough (0.99 accuracy, recall, precision and f1_score) for us to not need to tinker with predicted probabilities models.

Results

Figure 1 (below) contains confusion matrices for our final SVM and SGD models. Both of the final models had roughly 99.6% accuracy and this is evident in the confusion matrix. It's worth noting that the action "reset-both" accounts for only .08% of the data. There was not enough data to train on that action which reflects the poor performance for it.

Our final SVM model misclassified 41 out of 13,107 predictions or 0.3%.

Our final SGD model misclassified 42 out of 13,107 predictions or 0.3%.

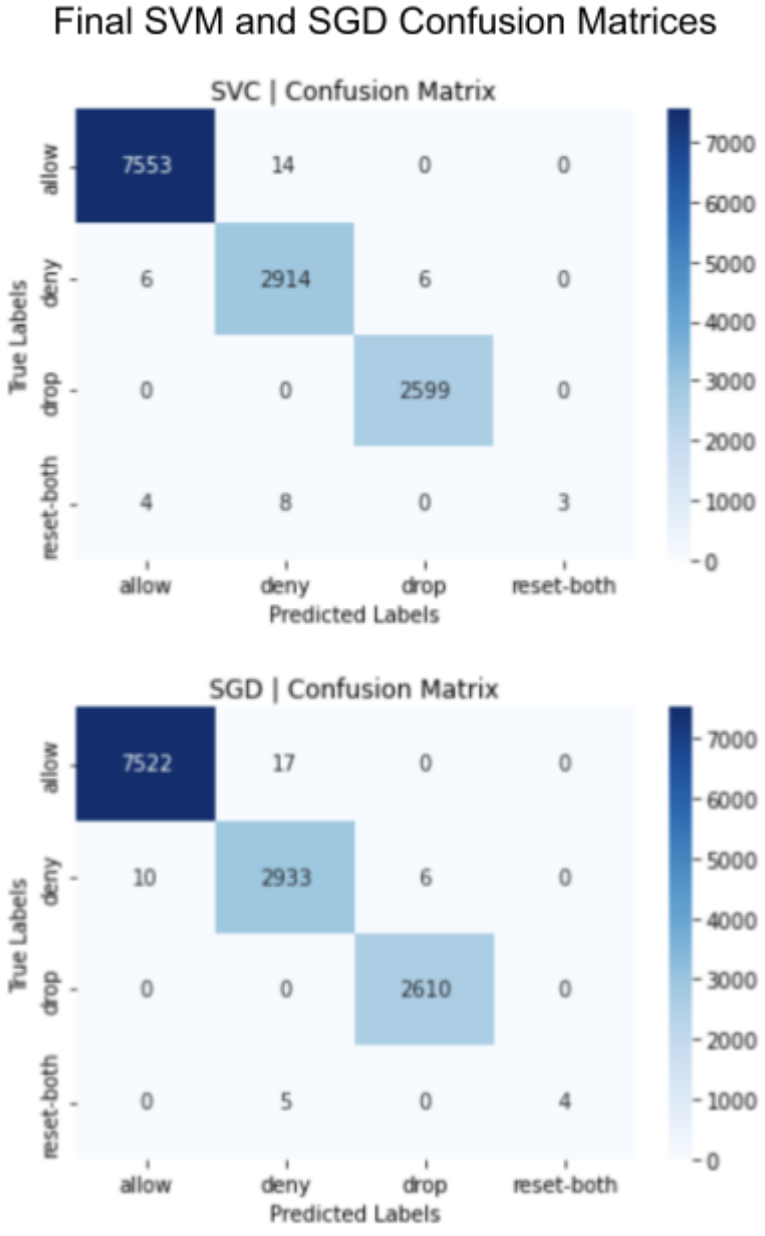

Figure 1 - SVM and SGD Confusion Matrices

Figure 2 (below) shows boxplots describing model results from fitting 5 SVM and 5 SGD models with a different train/test split each run. The orange lines within the boxes represent each metric's mean, while the box and whiskers represent each metric's variance. The smaller the variance, the more confident we can be in our model's consistency. As you can see, the variance for each metric for each of the models is very small, hinting that our models had consistent results.

For both our SVM and SGD models, we had between 99.6% and 99.7% accuracy, precision, recall, and f1_score over 5 runs. These models had means of 99.7% for each of these metrics.

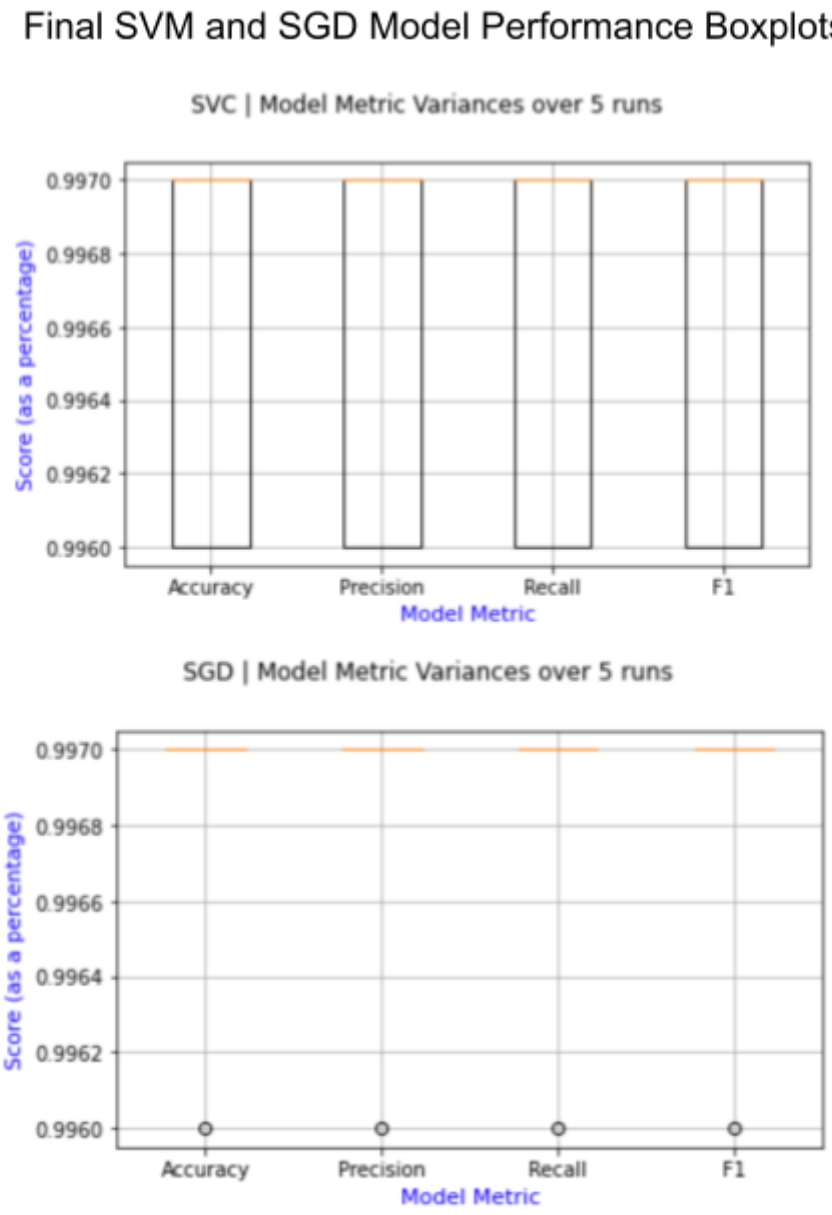

Figure 2 - SVM and SGD Model Performance Boxplots

Figure 3 (below) shows the mean metrics for our final models.

Our final SVM model received an average f1_score of 0.9966, recall of 0.9966, precision of 0.9966 and an accuracy of 0.9966.

Our final SGD model received an average f1_score of 0.9968, recall of 0.9968, precision of 0.9968, and an accuracy of 0.9968.

Our final SGD outperformed our final SVM by .0002 f1, recall, precision and accuracy. The SGD model also fit and made predictions more than twice as fast clock-time-wise as the SVM model.

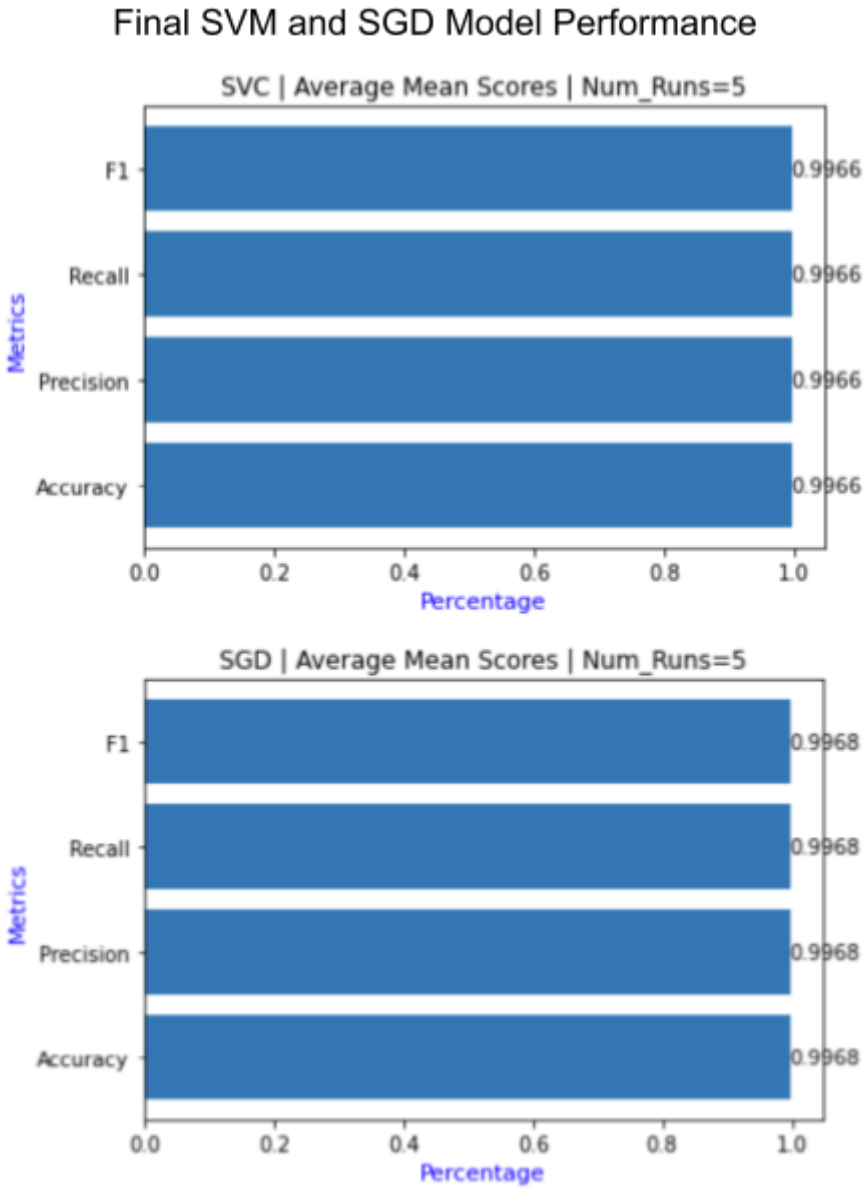

Figure 3 - SVM and SGD Model Performance

Below are descriptions of the results based on Figure 2 for our final models:

Accuracy: SVM: Mean = 99.66% | Variance = <1% | SGD: Mean = 99.68% | Variance = <1%

Accuracy is determined by the number of correctly predicted instances of the total number of instances. Our model had very high accuracy, and made very accurate predictions. We tuned our parameters around the model that returned the highest accuracy score. Our purpose behind this was because we were trying to get the highest accuracy possible because that was one of our objectives.

Precision: SVM: Mean = 99.66% | Variance = <1% | SGD: Mean = 99.68% | Variance = <1%

The equation for precision is: (true positives / (true positives / false positives)). The more false positives there are, the lower the precision score. In this project we focused on optimizing accuracy but still our final precision scores were extremely high. Our final SGD model outperformed our final SVM model by .02%.

Recall: SVM: Mean = 99.66% | Variance = <1% | SGD: Mean = 99.68% | Variance = <1%

Recall is similar to precision except it measures false negatives rates instead of false positive rates. The equation for recall is: (true positives / (true positives + false negatives)). In the context of this project, a lower recall would mean more data that would have been given the wrong action. Both of our models performed great, but the SGD model was .02% higher.

F1_Score: SVM: Mean = 99.66% | Variance = <1% | SGD: Mean = 99.68% | Variance = <1%

F1_score is the harmonic mean of precision and recall. The equation for f1_score is this: 2 * (precision * recall) / (precision + recall). The SGD model outperformed the SVM model overall in precision and in recall so the f1 was higher.

Feature importances

We could not get feature importances from our final SVM model because we did not use a linear kernel.

Here are the top 10 feature importances we received from our final SGD model:

Elapsed Time (sec) : 4.859063913291297
Destination Port_53 : 1.4146309941366513
Bytes : 1.3502980022178448
Bytes Received : 1.203414136276266
Destination Port_0 : 0.9511863432857917
Destination Port_8080 : 0.7037973830789287
Bytes Sent : 0.6941099971060313
Destination Port_60038 : 0.42046563455811947
Destination Port_22448 : 0.4176544517978673
Destination Port_161 : 0.30614981555863285

The top 10 features according to our SGD model were a mix of one-hot encoded categorical features and scaled continuous data. To our surprise, the number one most important feature, “Elapsed Time (sec)” was 3.44 times more important than the number two most important feature. This is a big jump compared to the jump from the top two and three, and so on.

Conclusion

Our SVM and SGD models performed very well. The mean accuracy for our final SVM model was 0.9966 and the mean accuracy for our final SGD model was 0.9968.

The parameters we used to create our final SVM model were: 'C'=1, 'kernel'='rbf', 'class_weight'=None, 'decision_function_shape'='ovr'.

The parameters we used to create our final SGD model were: 'loss'='modified_huber', 'penalty'='l2', 'alpha'=1e-6, 'max_iter'=1000, 'learning_rate'='optimal', 'early_stopping'=True, 'n_iter_no_change'=5, 'class_weight'=None.

Note: We realize max_iter=1000 is a default value for SGDClassifier() but we explicitly set it to 1000 in our final model params for readability.

Appendix A

Link to main script:
https://github.com/Abillelatus/QTW-Case-Studies/blob/main/CaseStudy_5/MSDS-7333-CaseStudy-5.py

Link to full CaseStudy_5 full contents:
https://github.com/Abillelatus/QTW-Case-Studies/tree/main/CaseStudy_5

Full list of columns:
Source Port: The port number on the sender's machine that is used to send the traffic.

Destination Port: The port number on the receiver's machine that is used to receive the traffic.

NAT Source Port: The network address translation (NAT) source port is a modified source port number that is used when network traffic is translated between different IP addresses or networks. NAT is often used in firewalls or routers to allow multiple devices to share a single IP address.

NAT Destination Port: The NAT destination port is a modified destination port number that is used when network traffic is translated between different IP addresses or networks.

Action: The action taken on the traffic by a network device or security system, such as allow, block, or alert.

Bytes: The total number of bytes in the network traffic.

Bytes Sent: The number of bytes sent by the source in the network traffic.

Bytes Received: The number of bytes received by the destination in the network traffic.

Packets: The total number of packets in the network traffic.

Elapsed Time (sec): The time duration for which the network traffic was observed.

pkts_sent: The number of packets sent by the source in the network traffic.

pkts_received: The number of packets received by the destination in the network traffic.