# Case Study V

Randy Kim, Kati Schuerger, Will Sherman
October 31, 2022

# 1 Introduction

## Objective

A large company with cybersecurity presented their concerns due to a high volume of firewall interactions. The goal was to use more advanced algorithms to build a program that will determine whether to accept or deny access within the firewall for requests that come in. Automating the current process with a high performing model will reduce manpower and a lot of resources.

The company provided historical data in which previous requests were either accepted or rejected. We used this data to create models that properly filter a large scale of incoming requests with high accuracy and efficiency. The focus of this study was on support vector machines and the optimization methods that could be employed to implement models for consideration at scale for the company.

# 2 Methods

## Computational Requirements

Final method and models were generated on a 4 CPU cluster with 48 cores (Intel® AVX2) with 90GB memory allocation.

## Data

The client provided historical data of how they've chosen to accept and deny these requests; about 60,000 rows (observations). Our group used this historical data to build a model that could be used to automatically filter future requests based on past information.

## Data pre-processing (EDA): Category alignment

The data provided by the client contained 4 options for the target category (Action): *accept*, *deny*, *dropped*, *reset-both*. The inquiry from the client was to build a model that will produce a binary outcome: to either accept or deny an incoming firewall request. To align with this request, we updated the data categories to include deny, drop, and reset-both, into the deny category (Table 2).

## Data pre-processing (EDA): Dimensionality reduction

We looked at the distribution of all ports in the original data and noted that there were multiple instances of port numbers with very few call-records. This overabundance of unique port-values made it difficult to identify a model that would fit this data. To reduce the dimensionality of the data we created a new category option: “rare” (Appendix: Figure 3).

Any records that had fewer than 5 calls in the data were re-categorized into this rare class per Port-type (*i.e.*, Source Port, Destination Port, NAT Source Port, NAT Destination Port).

## Classification Methods

We explored the support vector machine (SVM) classification algorithm and the stochastic gradient descent (SGD) optimization method for correctly identifying firewall action.

## Support vector machines

This machine learning method is an algorithm that tries to separate points by selecting a line or plane that separates two (or more) groups. SVM seeks to fashion the largest separation between observations and a plane. When data has many features, this plane is projected in multiple directions (hyperplane) if a linear solution is not possible for the data. The overall method to SVM is to find is the largest margin between these "linearly" separable classes.

## Useful kernels

There are several options of what type of kernel trick function to use, a few are listed here.

| Kernel | Mathematical form |
|---|---|
| linear | $(x, y)$ |
| polynomial | $(\gamma(x, y) + r)^d$ |
| RBF | $\exp(-\gamma\|x - y\|^2)$ |
| sigmoid | $\tanh(\gamma(x, y) + r)$ |

Table 1. Kernels that were evaluated for SVM classifier.

## Stochastic gradient descent

SGD is an optimization method that works more quickly than the SVM by itself. This is due to randomly selecting a smaller number of examples to find a gradient on which to optimize—approximating the loss as opposed to the SVM which seeks the exact loss.

## Model evaluation & Scoring metrics

We used F1 score for tuning and model evaluation: optimized for all SVM models under consideration in combination with 3-fold stratified cross-validation, whereas the SGD model was investigated using a 5-fold stratified cross-validation approach.

We also utilized a confusion matrix and a classification report to evaluate and compare each model. Both the matrix and the report provided measuring of Recall, Precision, and Accuracy. Depending on the models, it could be difficult to compare models with high precision and low recall or vice versa, so the classification report also provides F1-score (the harmonic mean of precision and recall) to measure them at the same time.

## Train/Test/Validate split

Before modeling, we randomized the arrangement of the data samples and separated the data into two groups: a training dataset for model fitting and a validation dataset (holdout group of 20%) that was used to measure model performance on unseen data—evaluating whether the model generalizes well. This was done with SKLearn's *train_test_split* with stratification dependent on the classes. Training data class distribution was confirmed post-split. Additional splitting of the

training data was performed to generate an approximately 10,000 record tuning set from which hyperparameter tuning was performed for the SVM kernels.

## Hyperparameter tuning

An important part of our model building process was tuning the parameters; we did not have prior knowledge of what these parameter values should be. The three primary kernels pursued within our SVM were *linear*, *poly*, and *rbf* using SKLearn's *SVC* method. All kernels were investigated using 3-fold cross-validation on a subset (20% of all training data) of the data and randomized search (SKLearn's *RandomizedSearchCV*) for hyperparameter tuning with F1 score.

The parameter space for SVC was 80 per kernel; we randomly searched 20 of these per kernel. Only 3-fold cross-validation was performed during tuning due to time scaling as the number of folds increased.

### Overfitting

One problem that needs to be guarded against when building a model is overfitting, this applies to SVM as well. Overfitting means that the model is biased to the training data and does not perform well when given new data. Regularization methods (Lasso or L1, Ridge or L2; the C parameter operates similarly to L2 for SVM) help to guard against this, by penalizing the model as it becomes more complex. As we project to higher and higher dimensions with SVM, we leveraged the regularization parameter C to prevent overfitting.

### C parameter

The C parameter is the regularization strength parameter, which helps guard against the model overfitting to the data. An overfit model is one that is biased to the training data and does not generalize well—meaning it has poor performance when new and unseen data is given to the model. We primarily focused on tuning this parameter for the full SVM method.

### Kernel

Our pipeline for SVM to identify the best model included the four kernels above. After initial tuning, we opted to omit sigmoid due to performance time (not completing within 2-hour window).

# 3 Results

The original dataset included multiple categories for actions performed, which did not fit the end-decision schema. These were re-classified to meet the decision boundary of *accept* or *deny* access within firewall (Table 2).

| Raw Data | |
|---|---|
| **Action** | **Counts** |
| Allow | 37,640 |
| Deny | 14,987 |
| Drop | 12,851 |
| Reset-both | 54 |

| Adjusted Data | |
|---|---|
| **Action** | **Counts** |
| Allow | 37,640 |
| Deny | 27,892 |

Table 2. (Left) Raw data and counts. (Right) Adjusted data to binary classification and counts.

A baseline was established for how much training time was required for each SVM (Table 3). This, plus analysis of the compute-time growth based on number of datapoints, drove our decision to further subset our data to do hyperparameter tuning on 10,000 records instead of the larger 50,000 record training set. Time scaling based on number of records was found to be linear at minimum (Appendix: Figure 4).

| Model | Training Time |
|---|---|
| Linear | 232.48 |
| Poly | 429.83 |
| RBF | 374.06 |

Table 3. Amount of training time per kernel for SVM. (10,000 record training subset)

The overall model performance for the linear kernel was under 99% on test data based on the F1 score; therefore, that model was discarded. The two SVM models for consideration are the poly and RBF kernels, with RBF attaining the highest F1 score (RBF: 99.84%, poly 99.78%). Classification rates can be seen in Figure 1. Of particular note is that the RBF SVM had higher precision (100%) at only classifying traffic which should be allowed as allowed.

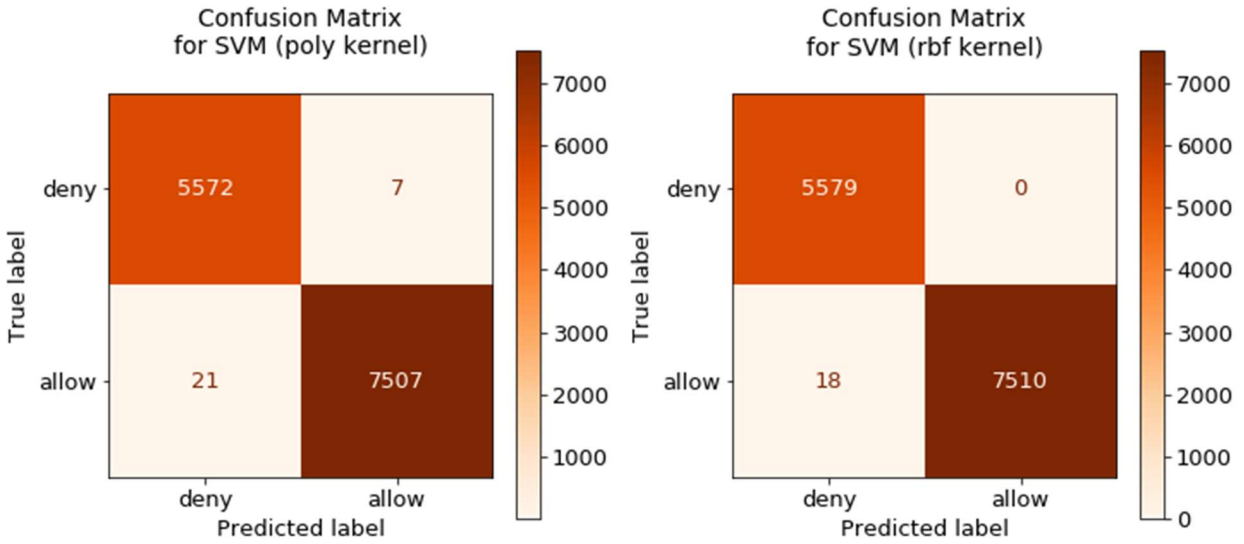


Figure 1. (a) The left confusion matrix shows correct classification rates for the poly kernel SVM. (b) The right confusion matrix shows correct classification by the RBF kernel SVM.

The RBF kernel also performed faster when training on the smaller tuning dataset; however, times were comparable as the size of the data increased.

During assessment, random splitting of the training and test data impacted the F1 score values. This was addressed by randomizing the random splitting of training and test data. In three splitting scenarios, the RBF kernel outperformed the poly kernel (Table 4). In fact, the lowest F1 score for RBF (0.9986) was still higher than the best score for the poly kernel (0.9983).

| | random_state | poly kernel | rbf kernel |
|---|---|---|---|
| **0** | 20 | 0.998271 | 0.998603 |
| **1** | 40 | 0.997937 | 0.998337 |
| **2** | 60 | 0.997739 | 0.998736 |

Table 4. F1 score as found during randomized splitting of train/test data. (Highest score per kernel hi-lighted)

The final method for consideration is the SGD classification method which utilizes the SVM approach but does not have time-scaling near the same order as the poly or RBF kernels. The results of the SGD classifier were comparable to the poly kernel with an F1 score of (99.75%) (Figure 2). The significant upside was that fitting this model against the entire 50,000 dataset occurred in under one minute (*caveat*, additional hyperparameter tuning was performed to optimize this; initial time investment still less than tuning regularization parameter, C, for SVMs).

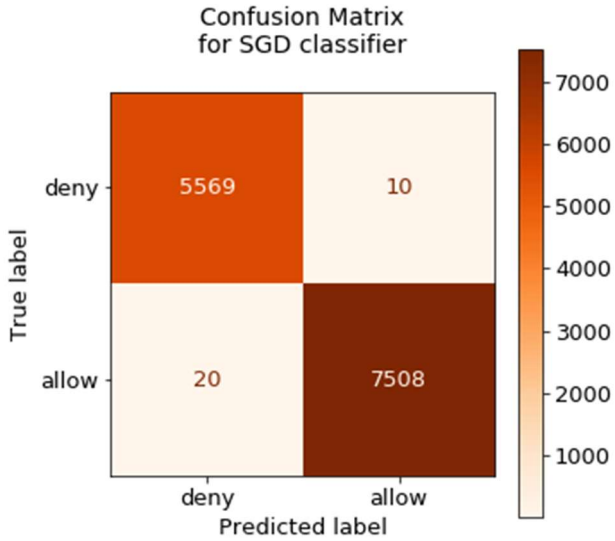


*Figure 2.* Confusion matrix showing correct classification rates for the SGD classifier.

# 4 Conclusion

The goal of this case study was to produce a model with high accuracy and precision that can secure the company's firewall. The SVM classifier produced an F1 score of 99.87% and the SGD classifier produced an F1 score of 99.75%. Although, compared to SGD classifier, the SVM produced a 0.12% higher accuracy score, it would be difficult to say one is better than others, based on F1 score, since all the models performed extremely similar with high accuracy and precision. However, if scalability is in question, our recommendation is to pursue the SGD classifier.

**Suggestions for next steps**

One possibility to improve the performance would be to build a separate model to handle ports that fall into the 'Rare' grouping. Because the representation for each port is so minimal, it might make sense to try clustering on these records, to seek alternative criteria to assist the model in understanding the inputs.

**Things to watch out for**

Scaling limitations of SVM are a hard limit as the scale of the data increases. If the same model is fit using a different dataset (more recent/current) or a decision is made to increase training data over time, SVM may not be a very useful method—given that it struggles to scale to larger inputs.

**N-squared scaling**

SVM is that it does not scale as well as some alternative algorithms. This is because the SVM must store the dot products (outputs from the kernel trick projecting to higher dimension), which are required to find the margin. The limit is storing these dot products in memory.

This is the origin of our n squared scaling; typically, SVM is not used once we reach about 50,000 rows of data (roughly the size of the data provided by the client in this case).

*Happy Halloween!*

# Appendix

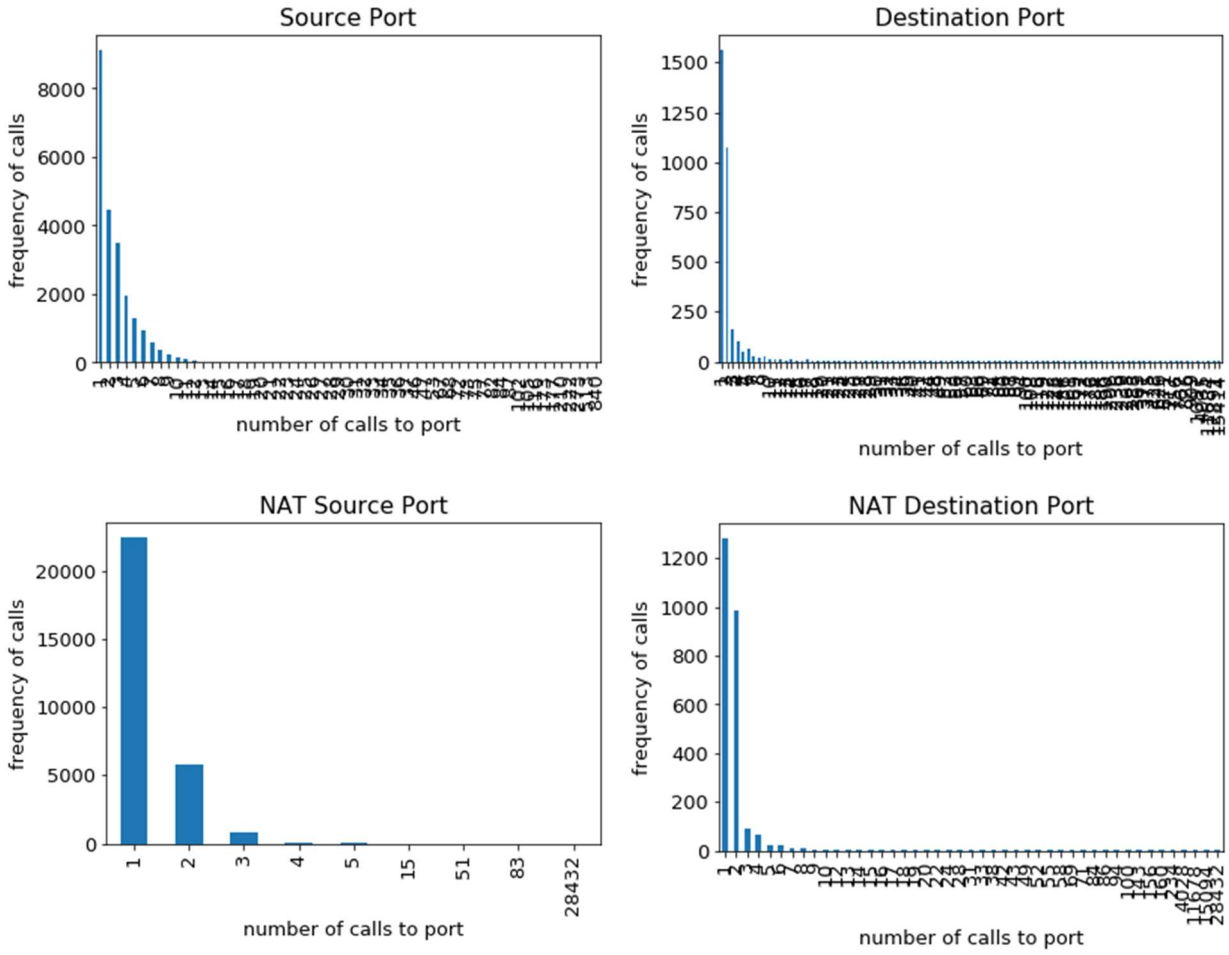


Figure 3. Distribution of Port calls based on number of calls made to different ports. High left-skew due to few calls.

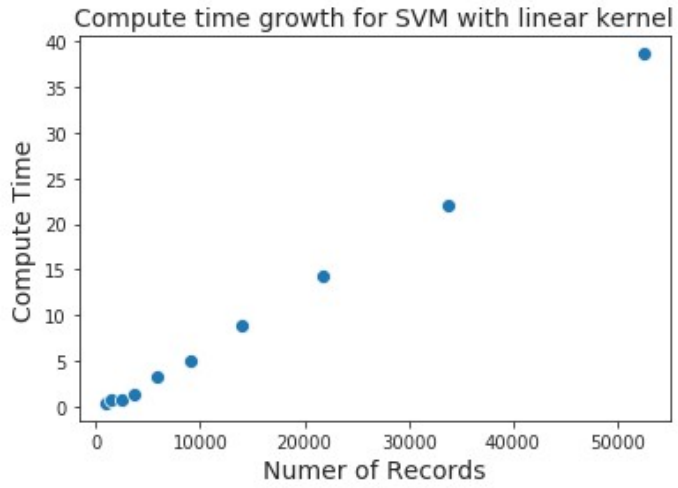


Figure 4. Linear growth for SVM (linear kernel). Highly undesirable for extremely large data.

# Code

```
import os
import numpy as np
import pandas as pd
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.model_selection import RandomizedSearchCV

from sklearn.svm import LinearSVC
from sklearn.svm import SVC
from sklearn.linear_model import SGDClassifier
from sklearn.metrics import ConfusionMatrixDisplay, confusion_matrix

data_path = 'log2.csv'
log2_data = pd.read_csv(data_path)
log2_data

for col in log2_data.columns: #convert port columns to string
    if('Port' in col):
        log2_data[col] = log2_data[col].astype(str)

type(log2_data['NAT Destination Port'][0])

log2_data.columns

log2_OHE = log2_data.copy()
log2_OHE

port_list = ['Source Port','Destination Port','NAT Source Port','NAT Destination Port']
num_vectors = [x for x in log2_OHE.columns if x not in port_list]

import matplotlib.pyplot as plt

eval_bins = [0,5,10,50,100,200,500,100,999999]
for port in port_list:
    plotter = log2_OHE[port].value_counts()
    fig, ax = plt.subplots()
    plotter.value_counts().sort_index().plot(ax = ax, kind='bar')
    plt.title(port)
    ax.set_xlabel('number of calls to port')
    ax.set_ylabel('frequency of calls')
    plt.show()

for port in port_list:
    mask = log2_OHE[port].map(log2_OHE[port].value_counts()) < 5
    log2_OHE[port] = log2_OHE[port].mask(mask, 'rare')

log2_OHE

log2_OHE = pd.get_dummies(log2_OHE, columns=port_list, drop_first=True)

log2_OHE
```

```
del num_vectors[0]
num_vectors

x_data = log2_OHE.iloc[:, 1:]
y_data = log2_OHE.iloc[:, 0]

x_data

y_data.value_counts()

y_data.loc[y_data != 'allow'] = 'deny'
y_data.value_counts()

# Setting allow as positive case
y_data.loc[y_data == 'allow'] = 1
y_data.loc[y_data == 'deny'] = 0
x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, test_size=0.2, stratify=y_data, random_state=192)

print(x_train.shape)

scale = np.logspace(3, np.log10(x_train.shape[0]), 10)
scale = [int(item) for item in scale]
scale

import time

time_logger = {}
for i in scale:
    svc = LinearSVC(max_iter=50000)
    begin = time.time()
    svc.fit(x_train.iloc[:i, :], y_train.iloc[:i])
    time_logger[i] = (time.time() - begin)

from pprint import pprint
pprint(time_logger)

import seaborn as sns
df = pd.DataFrame(time_logger.items(), columns=['datapoints','time'])
sns.set_style('ticks')
sns.scatterplot(data=df, x='datapoints', y='time', s=64)
plt.title('Compute time growth for SVM with linear kernel', fontsize=14)
plt.xlabel('Numer of Records',fontsize=14)
plt.ylabel('Compute Time',fontsize=14)

from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
num_vectors

x_train[num_vectors] = scaler.fit_transform(x_train[num_vectors])
x_train

x_train.shape

x_tune_set, x_remainder, y_tune_set, y_remainder = train_test_split(x_train, y_train, train_size=0.2,
                                                    stratify=y_train, random_state=12)
```

```
x_tune_set.shape

y_tune_set.value_counts()

y_test.value_counts()

x_test[num_vectors] = scaler.fit_transform(x_test[num_vectors])
x_test

# # SVM Section: Don't Run unless on HPC

kernels = ["linear","poly", "rbf", "sigmoid"]
svcs = [SVC(kernel=k, cache_size=8000) for k in kernels]
model_dict = {k:v for k,v in zip(kernels,svcs)}

time_logger = {}
for v in model_dict.values():
   begin = time.time()
   v.fit(x_tune_set, y_tune_set)
   time_logger[v] = (time.time() - begin)
   print(time_logger[v])


params = {'C': np.logspace(-5,5,40),
        'class_weight': [None, 'weighted'],
        'random_state': [415]}

del model_dict['sigmoid'] #dropping poly due to time constraints

model_dict

param_tune = {}
begin = time.time()
for kernel, mdl in model_dict.items():
   g_search = RandomizedSearchCV(mdl,
                        param_distributions=params,
                        n_iter=20,
                        n_jobs=48,
                        cv=3,
                        random_state=903,
                        scoring='f1')

   g_search.fit(x_tune_set, y_tune_set)
   param_tune[mdl] = [g_search.best_estimator_,
                g_search.best_score_,
                g_search.cv_results_]

   print(kernel, ' finished in ', g_search.cv_results_['mean_score_time'].sum().round(2))

print('total time to tune models', round((time.time()-begin),2))

param_tune

from tabulate import tabulate
```

```
print(tabulate([['Model','Training Time'],
                ['Linear',232.48],
                ['Poly',429.83],
                ['RBF',374.06]],
               headers='firstrow',
               tablefmt='fancy_grid'))

# this prints the model so we don't need to run this multiple times
with open('best_SVMs.txt', 'w') as f:
    for key, value in param_tune.items():
        f.write('%s:%s\n'%(key, value))

tuned_SVCs = []
i=0

for k in param_tune:
    tuned_SVCs.append(param_tune[k][0])
    print('F1 score for {} was {}'.format(list(model_dict.keys())[i], param_tune[k][1].round(5)))
    i += 1

tuned_SVCs

y_preds = []

for j, svc in enumerate(tuned_SVCs):
    begin = time.time()
    svc.fit(x_train, y_train)
    y_preds.append(svc.predict(x_test))
    print('done with model {} in {}'.format(j+1,round(time.time()-begin,2)))

print('time to train & predict with tuned models',time.time() - begin)

y_preds[0]

cm = confusion_matrix(y_test, y_preds[0])

font = {'size'   : 13}
plt.rc('font', **font)

c_disp = ConfusionMatrixDisplay(cm, display_labels=np.asarray(['deny','allow']))
fig, ax = plt.subplots(figsize=(5,5))
ax.grid(False)
plt.title('Confusion Matrix\nfor SVM (linear kernel)\n', fontsize=14)
c_disp.plot(cmap=plt.cm.Oranges, ax=ax, values_format='')

cm = confusion_matrix(y_test, y_preds[1])

font = {'size'   : 13}
plt.rc('font', **font)

c_disp = ConfusionMatrixDisplay(cm, display_labels=np.asarray(['deny','allow']))
fig, ax = plt.subplots(figsize=(5,5))
ax.grid(False)
plt.title('Confusion Matrix\nfor SVM (poly kernel)\n', fontsize=14)
c_disp.plot(cmap=plt.cm.Oranges, ax=ax, values_format='')
```

```
cm = confusion_matrix(y_test, y_preds[2])

font = {'size'   : 13}
plt.rc('font', **font)

c_disp = ConfusionMatrixDisplay(cm, display_labels=np.asarray(['deny','allow']))
fig, ax = plt.subplots(figsize=(5,5))
ax.grid(False)
plt.title('Confusion Matrix\nfor SVM (rbf kernel)\n', fontsize=14)
c_disp.plot(cmap=plt.cm.Oranges, ax=ax, values_format='')
```

**# ## Add this to show Random State issues**

```
rands = np.linspace(20,80,3)
rands = [int(x) for x in rands]
x_train_lists = []
x_test_lists = []
y_train_lists = []
y_test_lists = []

for r in rands:
    x_Tr, x_Te, y_Tr, y_Te = train_test_split(x_data, y_data, test_size=0.2, stratify=y_data, random_state=r)
    x_Tr[num_vectors] = scaler.fit_transform(x_Tr[num_vectors])
    x_Te[num_vectors] = scaler.fit_transform(x_Te[num_vectors])

    x_train_lists.append(x_Tr)
    x_test_lists.append(x_Te)
    y_train_lists.append(y_Tr)
    y_test_lists.append(y_Te)

len(x_train_lists)

from sklearn.metrics import f1_score
poly_F1 = []
rbf_F1 = []

for i in range(len(x_train_lists)):
    begin = time.time()
    poly = tuned_SVCs[1]
    rbf = tuned_SVCs[2]
    poly.fit(x_train_lists[i], y_train_lists[i])
    rbf.fit(x_train_lists[i], y_train_lists[i])

    poly_preds = poly.predict(x_test_lists[i])
    rbf_preds = rbf.predict(x_test_lists[i])

    poly_F1.append(f1_score(y_test_lists[i], poly_preds))
    rbf_F1.append(f1_score(y_test_lists[i], rbf_preds))
    print('Done with random set ', i+1, 'in ', (time.time()-begin))

rbf_F1

f1_rands = pd.DataFrame.from_dict({'poly kernel': poly_F1,
                                   'rbf kernel': rbf_F1})
```

```
max_rands = f1_rands.max()
print(max_rands)

def highlight_max(s):
   is_max = s == s.max()
   return ['background: #ffff7d' if cell else '' for cell in is_max]

pd.set_option('precision',6)
f1_rands.style.apply(highlight_max)

# # SGD Classifier Section

sgd_hyps = {'loss':['hinge', 'modified_huber', 'squared_hinge', 'perceptron',
                'huber', 'epsilon_insensitive', 'squared_epsilon_insensitive'],
         'alpha':np.logspace(-8,8,100),
         'fit_intercept':[True, False],
         'tol':np.logspace(-8,-2,7),
         'epsilon':np.logspace(-4,4,9),
         'learning_rate':['optimal','constant','invscaling','adaptive'],
         'eta0':np.logspace(-3,9,13),
         'power_t':[x + 0.5 for x in np.linspace(-10000,10000,21)],
         'validation_fraction':[0.2],
         'n_iter_no_change':[5],
         'class_weight':[None,'balanced']
}

sgdc = SGDClassifier(max_iter=10000, penalty='l2', fit_intercept=True, n_jobs=48, random_state=162)

gd_search = RandomizedSearchCV(sgdc,
                        param_distributions=sgd_hyps,
                        n_iter=100,
                        n_jobs=48,
                        cv=5,
                        random_state=111,
                        scoring='f1')

gd_search.fit(x_tune_set, y_tune_set)

best_sgdc = gd_search.best_estimator_
print('f1 score:\t\t\t',gd_search.best_score_)
print('time to fit individual model:\t',gd_search.cv_results_['mean_score_time'].sum().round(2))
print(best_sgdc,'\n')
print(gd_search.cv_results_)

begin = time.time()
best_sgdc.fit(x_train, y_train)
sgdc_preds = best_sgdc.predict(x_test)
print('time to train & predict with tuned model',time.time() - begin)

best_sgdc.score(x_train, y_train) #checking internal scoring against 5-fold scoring done during hyperparameter
tuning

cm = confusion_matrix(y_test, sgdc_preds)
```

```
font = {'size'   : 13}
plt.rc('font', **font)

c_disp = ConfusionMatrixDisplay(cm, display_labels=np.asarray(['deny','allow']))
fig, ax = plt.subplots(figsize=(5,5))
ax.grid(False)
plt.title('Confusion Matrix\nfor SGD classifier\n', fontsize=14)
c_disp.plot(cmap=plt.cm.Oranges, ax=ax, values_format='')
```