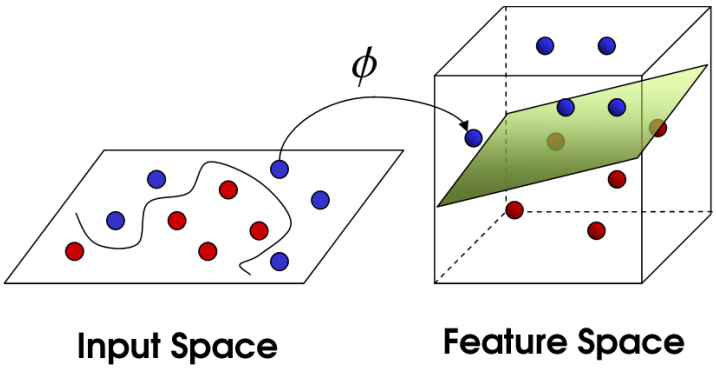
Support Vector Machines

Comparison between SVM, k-nearest Neighbour and Nearest Class Centroid

Iasonas Kakandris



A report presented for the course of
Neural Networks: Deep learning

Aristotle University of Thessaloniki
Faculty of Electrical and Computer Engineering
4/12/2023

Contents

1 Introduction

The goal of this project is to test the ability of a Support Virtual Machine (COTS or in-house) to correctly categorize photos from the CIFAR-10 dataset. More specifically we will use part of the dataset to train SVMs from the sklearn library (10,000 training data and 2,000 test data). This decision was made due to the high value of data in the dataset which makes the training really slow and cannot provide results. Furthermore, we will use 2 of the 10 categories (again 10,000 training data 2,000 test data) to train both COTS SVMs and our in-house SVMs in order to compare them. In the end, we will compare the accuracy of the aforementioned SVMs with the accuracy of the K-Nearest Neighbor and Centroid Neighbor algorithm to find the best results. Also we used Neptune AI to check the accuracy and other metrics for our SVM.

2 Code description

In order to test multiple SVMs we used the sklearn library which has COTS SVMs, as well as an in-house SVM. To understand the code behind the project, this section consists of an in death description of code that was developed

2.1 How to create Sections and Subsections

Simply use the section and subsection commands, as in this example document! With Overleaf, all the formatting and numbering is handled automatically according to the template you've chosen. If you're using the Visual Editor, you can also create new section and subsections via the buttons in the editor toolbar.

2.2 Load and Normalize the CIFAR-10 dataset

In order to load the CIFAR-10 dataset we will use the 'tensorflow.keras.datasets' library and import and 'cifar10'. Secondly we will load the training and test datasets in our code and normalize them by dividing them with 255. This happens because our images have rgb values which range from 0 to 255. In order to normalize them we divide the data with the maximum possible values they can have, which is 255. Then we initialize our categorize and reshape the 'y_train' and 'y_test' variables which include our categorize from 1 column with multiple rows, to 1 row with multiple columns. Next we check if an image can be loaded with the correct category by creating a function 'showImage()' and finally we shuffle our dataset in order to have better distribution of our dataset.

```
#%% CIFAR 10 dataset

# Load CIFAR-10 dataset
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

# make the rgb value from 0 - 255 --> 0 - 1 ==> scaling
X_train, X_test = X_train / 255.0, X_test / 255.0

# CIFAR-10 class names
class_names = ['airplane', 'automobile', 'bird', 'cat',
               'deer', 'dog', 'frog', 'horse', 'ship', 'truck']


# from 1 col mul rows --> 1 row mul cols
print(y_train)
y_train = y_train.reshape(-1,)
print(y_train)

print(y_test)
y_test = y_test.reshape(-1,)
print(y_test)

# test to see if it works properly


def showImage(x, y, index):
    plt.figure(figsize=(15, 2))
    plt.imshow(x[index])
    plt.xlabel(class_names[y[index]])


showImage(X_train, y_train, random.randint(0, 9))

# the train and test data
print(X_train.shape, X_test.shape)
```

```
# Shuffle the training dataset
keys = np.array(range(X_train.shape[0]))
np.random.shuffle(keys)
X_train = X_train[keys]
y_train = y_train[keys]
```

2.3 Reshape data

This part of the code reshapes the original image data from a 4D array to a 2D array. It extracts information about the number of samples, image height, image width, and the number of channels (e.g., RGB channels) from the shape of the X_train array. The image data is then flattened, converting each image into a one-dimensional vector. The resulting shape is (num_samples, img_height * img_width * num_channels).

Note: This part of the code is commented out in our project but is the general rule we use to reshape our data. For that we won't re-explain the similar procedure happening again later in our code

```
# we want to reshape the image from a 4D array to a 2D array
# This line extracts the number of samples, image height, image width, and number of
    channels (e.g., RGB channels) from the shape of the X_train array.
num_samples, img_height, img_width, num_channels = X_train.shape

# it flattens the image data, converting each image into a one-dimensional vector.
# The resulting shape is (num_samples, img_height * img_width * num_channels),
X_train = X_train.reshape(num_samples, -1)
num_samples, img_height, img_width, num_channels = X_test.shape
X_test = X_test.reshape(num_samples, -1)
```

2.4 Choosing Categories

This part of the code selects two classes (e.g., 'airplane' and 'automobile') based on their class indices and filters the training and test data accordingly. It prints the shape of the resulting filtered datasets to provide information about the number of samples and features for both training and test sets.

```
#%% Select two classes (e.g., 'airplane' and 'automobile')

class1, class2 = 0, 1 # You can choose the class indices based on the CIFAR-10 class names

# Filter training data and labels for the selected classes
selected_train_indices = np.where((y_train == class1) | (y_train == class2))[0]
X_train_selected = X_train[selected_train_indices]
y_train_selected = y_train[selected_train_indices]

# Filter test data and labels for the selected classes
selected_test_indices = np.where((y_test == class1) | (y_test == class2))[0]
X_test_selected = X_test[selected_test_indices]
y_test_selected = y_test[selected_test_indices]


# Print the shape of the filtered datasets
print("Shape of filtered training data:", X_train_selected.shape)
print("Shape of filtered training labels:", y_train_selected.shape)
print("Shape of filtered test data:", X_test_selected.shape)
print("Shape of filtered test labels:", y_test_selected.shape)
```

2.5 SVMs with different kernels and 2 categories of data

For our first experiment, we will use multiple SVMs with different kernels in order to find the best one for categorizing the classes. We will compare their accuracy as well as the time needed for each one to conclude. In order to have similar comparisons we will use C=1. The kernels we use are:

- sigmoid
- polynomial
- radial basis function
- linear

We will also show the confusion matrix for each one of them to have a better understanding of the results

---

```
#%% Sigmoid kernel

# instantiate classifier with sigmoid kernel and C=1.0
sigmoid_svc=SVC(kernel='sigmoid', C=1.0)


# fit classifier to training set
sigmoid_svc.fit(X_train_selected,y_train_selected)

# make predictions on test set
y_pred=sigmoid_svc.predict(X_test_selected)


# compute and print accuracy score
print('Model accuracy score with sigmoid kernel and C=1.0 : {0:0.4f}'.
    format(accuracy_score(y_test_selected, y_pred)))

#Model accuracy score with sigmoid kernel and C=1.0 : 0.6925

# visualize confusion matrix with seaborn heatmap
cm = confusion_matrix(y_test_selected, y_pred)
class_names_selected = ['airplane', 'automobile']


heatmap_1 = sns.heatmap(
     cm, annot=True, fmt='d', cmap='YlGnBu', xticklabels=class_names_selected,
         yticklabels=class_names_selected)
heatmap_1.set_title("confusion matrix for neural network TEST")
plt.xlabel('Predicted')
plt.ylabel('Actual')
 # print(classification_report(batch_y, output_for_matrix))
plt.show()


#%% Polynomial kernel

# instantiate classifier with polynomial kernel and C=1.0
poly_svc=SVC(kernel='poly', C=1.0)
# degree int, default=3

# fit classifier to training set
poly_svc.fit(X_train_selected,y_train_selected)


# make predictions on test set
```

```
y_pred=poly_svc.predict(X_test_selected)


# compute and print accuracy score
print('Model accuracy score with polynomial kernel and C=1.0 : {0:0.4f}'.
    format(accuracy_score(y_test_selected, y_pred)))

#Model accuracy score with polynomial kernel and C=1.0 : 0.9145

#%% Confusion matrix

# visualize confusion matrix with seaborn heatmap
cm = confusion_matrix(y_test_selected, y_pred)
class_names_selected = ['airplane', 'automobile']


heatmap_1 = sns.heatmap(
     cm, annot=True, fmt='d', cmap='YlGnBu', xticklabels=class_names_selected,
         yticklabels=class_names_selected)
heatmap_1.set_title("confusion matrix for neural network TEST")
plt.xlabel('Predicted')
plt.ylabel('Actual')
 # print(classification_report(batch_y, output_for_matrix))
plt.show()

#sns.heatmap(cm_matrix, annot=True, fmt='d', cmap='YlGnBu')

#%% Default hyperparameter means C=1.0, kernel=rbf and gamma=auto among other parameters.

# instantiate classifier with default hyperparameters
svc=SVC()


# fit classifier to training set
svc.fit(X_train_selected,y_train_selected)

# make predictions on test set
y_pred=svc.predict(X_test_selected)


# compute and print accuracy score
print('Model accuracy score with default hyperparameters: {0:0.4f}'.
    format(accuracy_score(y_test_selected, y_pred)))
#Model accuracy score with default hyperparameters: 0.9040

cm = confusion_matrix(y_test_selected, y_pred)
class_names_selected = ['airplane', 'automobile']

heatmap_1 = sns.heatmap(
     cm, annot=True, fmt='d', cmap='YlGnBu', xticklabels=class_names_selected,
         yticklabels=class_names_selected)
heatmap_1.set_title("confusion matrix for neural network TEST")
plt.xlabel('Predicted')
plt.ylabel('Actual')
 # print(classification_report(batch_y, output_for_matrix))
plt.show()

#%% linear kernel


# instantiate classifier with polynomial kernel and C=1.0
linear_svc=SVC(kernel='linear')
```

```
# degree int, default=3

# fit classifier to training set
linear_svc.fit(X_train_selected,y_train_selected)


# make predictions on test set
y_pred=linear_svc.predict(X_test_selected)


# compute and print accuracy score
print('Model accuracy score with polynomial kernel and C=1.0 : {0:0.4f}'.
    format(accuracy_score(y_test_selected, y_pred)))

#Model accuracy score with polynomial kernel and C=1.0 : 0.9145

cm = confusion_matrix(y_test_selected, y_pred)
class_names_selected = ['airplane', 'automobile']

heatmap_1 = sns.heatmap(
    cm, annot=True, fmt='d', cmap='YlGnBu', xticklabels=class_names_selected,
        yticklabels=class_names_selected)
heatmap_1.set_title("confusion matrix for neural network TEST")
plt.xlabel('Predicted')
plt.ylabel('Actual')
 # print(classification_report(batch_y, output_for_matrix))
plt.show()
```

2.6 SVMs using GridSearch with batched data from all categories

The batched data will not have exactly equal number of images from each category which may make our results a bit optimized for the category with the most data.

2.6.1 GridSearchCV Setup

GridSearchCV is imported from sklearn.model_selection. It is a technique for hyperparameter tuning that exhaustively searches a specified parameter grid to find the best combination of hyperparameters for a given model.

2.6.2 Best Model Analysis

The code prints the best cross-validated score achieved during the grid search (grid_search.best_score_), the parameters that resulted in the best score (grid_search.best_params_) and the best estimator (SVM model) chosen by the grid search (grid_search.best_estimator_). Test Data Evaluation:

The code calculates and prints the accuracy score on the test set (X_batch_test, y_batch_test) using the best model found during the grid search.

2.6.3 Parameter Grid

A parameter grid (parameters) is declared for hyperparameter tuning. It includes different values for the regularization parameter C, kernel type, degree (for polynomial kernel), and gamma.

Note: Due to the high number of different combinations, it was impossible to run by using the CIFAR-10 dataset. It was tested with other Datasets and it was working correctly. Furthermore it was used for linear kernel with c = 1 and 10 and a result was achieved after $\approx 40 mins$. An attempt at a simplified version can be seen being used in Section 4.2.2

```
#%% Hyperparameter Optimization using GridSearch CV

# import GridSearchCV
from sklearn.model_selection import GridSearchCV


# import SVC classifier
from sklearn.svm import SVC


# instantiate classifier with default hyperparameters with kernel=rbf, C=1.0 and gamma=auto
svc=SVC()



# declare parameters for hyperparameter tuning
parameters = [ {'C':[1, 10], 'kernel':['linear']},
              {'C':[1, 10, 100, 1000], 'kernel':['rbf'], 'gamma':[0.1, 0.2, 0.3, 0.4, 0.5,
                  0.6, 0.7, 0.8, 0.9]},
              {'C':[1, 10, 100, 1000], 'kernel':['poly'], 'degree': [2,3,4]
                  ,'gamma':[0.01,0.02,0.03,0.04,0.05]}
             ]


grid_search = GridSearchCV(estimator = svc,
                           param_grid = parameters,
                           scoring = 'accuracy',
                           cv = 5,
                           verbose=0)

#%% fit the data
grid_search.fit(X_batch, y_batch)

#%% best model

# examine the best model


# best score achieved during the GridSearchCV
print('GridSearch CV best score : {:.4f}\n\n'.format(grid_search.best_score_))


# print parameters that give the best results
print('Parameters that give the best results :','\n\n', (grid_search.best_params_))


# print estimator that was chosen by the GridSearch
print('\n\nEstimator that was chosen by the search :','\n\n', (grid_search.best_estimator_))

#%% test data in gridsearch
# calculate GridSearch CV score on test set

print('GridSearch CV score on test set: {0:0.4f}'.format(grid_search.score(X_batch_test,
    y_batch_test)))
```

2.7 In-house SVM with 2 categories of data

This code defines a simple Support Vector Machine (SVM) with linear kernel class from scratch based on this youtube video: [Ass]. The training process is a simplified version of the perceptron learning

algorithm for linear SVMs. It aims to find a hyperplane that separates the data points of different classes while penalizing misclassifications. The regularization term (lambda_param) helps control overfitting by discouraging overly complex models

2.7.1 Initialization

The __init__ method initializes the SVM instance with hyperparameters such as learning rate (learning_rate), regularization parameter (lambda_param), and the number of iterations (n_iters). It also initializes the weights and bias to None.

2.7.2 Fit Method

The fit method trains the SVM on the input data (X, y). It uses the perceptron learning algorithm to update weights and biases iteratively. The labels y are converted to -1 and 1. The training loop runs for the specified number of iterations, and for each iteration, it updates weights and bias based on whether a certain condition is met.

2.7.3 Prediction Method

The predict method takes input data (X) and calculates an approximation using the learned weights and bias. The predicted labels are determined by taking the sign of the approximation.

2.7.4 Accuracy Logging

Within the training loop, accuracy during training is calculated using the accuracy_score function from scikit-learn. The accuracy values are appended to a list (run["accuracy"]) for later analysis or visualization.

```
#%% self made svm

class SVM_from_scratch :

    def __init__(self , learning_rate = 0.001 , lambda_param = 0.01 , n_iters = 1000):
        self.learning_rate = learning_rate
        self.lambda_param = lambda_param
        self.n_iters = n_iters
        self.weights = None
        self.bias = None

    def fit(self , X , y):
        n_samples , n_features = X.shape

        y_ = np.where(y <= 0 , -1 , 1)

        #init weights
        self.weights = np.zeros(n_features)
        self.bias = 0

        for _ in range(self.n_iters):
            for index , x_i in enumerate(X):
                condition = y_[index] * (np.dot(x_i , self.weights) - self.bias) >= 1

                if condition:

                    self.weights -= self.learning_rate * (2 * self.lambda_param *
                        self.weights) # a -> learning rate

                else:
```

```
                self.weights -= self.learning_rate * (2 * self.lambda_param * self.weights
                    - np.dot(x_i , y_[index]))
                self.bias -= self.learning_rate * y_[index]

        predicted = self.predict(X)
        accuracy = accuracy_score(y, predicted)
        run["accuracy"].append(accuracy)

def predict(self , X):
    approx = np.dot(X , self.weights) - self.bias

    return np.sign(approx)
```

3 Issues

3.1 Size of the data

The first issue that was encountered was the sheer size of the data. The 50,000 training and 10,000 test data could not be processed in order to have a final result. Multiple tests on different computers were done to reach a result but to no end. Furthermore we tried allowing the program to run on the GPU for faster results but the instructions found on the web were not helpful and the idea was ultimately abandoned. To surpass this problem 2 methods were selected with the same size of data: the selection and comparison of only to classes instead of all and the selection of a batch of the data. The batch was the size of 10,000 (same as the size of 2 categories) which made it easier for the computer to run the calculations.

3.2 In-house SVM algorithm

Our in-house SVM is classifying by using the below conversion

$$y = \begin{cases} -1 & y \leq 0 \\ 1 & y \geq 0 \end{cases}$$

Our data though have classification names (e.g airplane, automobile). We need to convert these values to -1 and 1 in order to use the SVM.

Note: For multiclass classification we could use the *One to rest approach* described in baeldung site by modifying the code.

```
#%% prepare dataset for the self made svm

# Convert class names to numeric labels in y_train_selected and y_test_selected
y_train_selected_numeric = np.where(y_train_selected == class1, -1, 1)
y_test_selected_numeric = np.where(y_test_selected == class1, -1, 1)

# Print the shape of the filtered datasets
print("Shape of filtered training data:", X_train_selected.shape)
print("Shape of filtered training labels:", y_train_selected_numeric.shape)
print("Shape of filtered test data:", X_test_selected.shape)
print("Shape of filtered test labels:", y_test_selected_numeric.shape)

# we want to reshape the image from a 4D array to a 2D array
# This line extracts the number of samples, image height, image width, and number of
    channels (e.g., RGB channels) from the shape of the X_train array.
num_samples, img_height, img_width, num_channels = X_train_selected.shape

# it flattens the image data, converting each image into a one-dimensional vector.
# The resulting shape is (num_samples, img_height * img_width * num_channels),
X_train_selected = X_train_selected.reshape(num_samples, -1)
num_samples, img_height, img_width, num_channels = X_test_selected.shape
X_test_selected = X_test_selected.reshape(num_samples, -1)
```

3.3 GridSearch for classification

As previously mentioned in section 2.6 the size of the data (even when batched) was making the GridSearch function unusable. Although we were unable to use it for the purpose of this project, it is mentioned because of its convenience and could be used in better-performance computers. An attempt of the usage of a simplified version of it can be seen in section 4.2.2

4 Results

4.1 2 classes classification

As we mention in section 2.5 In order to find the best SVM for the 2 classes classification we use the same C = 1.

4.1.1 Linear kernel

On the linear kernel the accuracy score that was achieved was : **0.80**
The time that was needed to be achieved was: **199.87** seconds
The classification report is in table 1

Table 1: Classification Report

	Precision	Recall	F1-Score	Support
0	0.78	0.83	0.81	1000
1	0.82	0.77	0.79	1000
Accuracy			0.80	2000
Macro Avg	0.80	0.80	0.80	2000
Weighted Avg	0.80	0.80	0.80	2000

The confusion matrix is in Figure 1

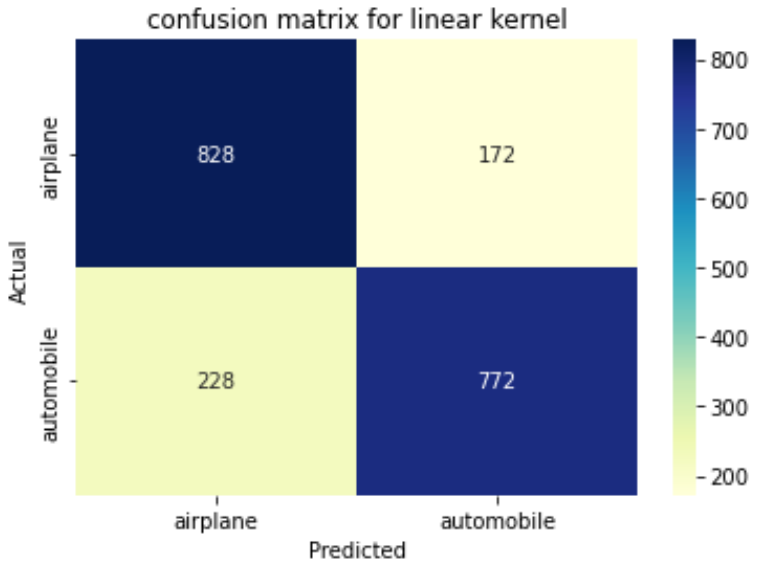


Figure 1: Confusion matrix for linear kernel

We can see that the Linear kernel is not as good as we wanted for a classifier and its run time is really slow.

4.1.2 Polynomial kernel

On the Polynomial kernel, the accuracy score that was achieved was: **0.9145**
The time that was needed to be achieved was: **76.44** seconds
The classification report can be seen in table 2
The confusion matrix can be seen in Figure 2
We conclude that the polynomial kernel is really good at classifying the 2 types of images.

Table 2: Classification Report for polynomial kernel

	Precision	Recall	F1-Score	Support
airplane	0.91	0.92	0.91	1000
automobile	0.92	0.91	0.91	1000
Accuracy			0.91	2000
Macro Avg	0.91	0.91	0.91	2000
Weighted Avg	0.91	0.91	0.91	2000

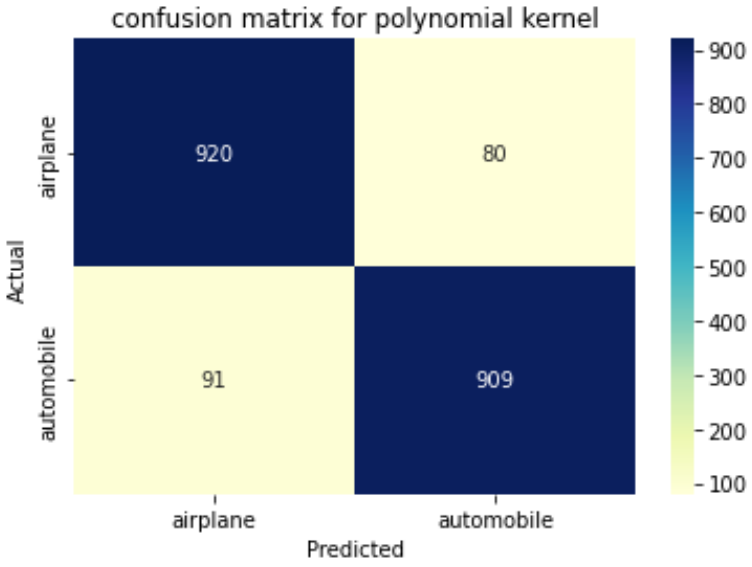


Figure 2: Confusion matrix for polynomial kernel

4.1.3 Radial basis function kernel

On the Radial basis function kernel the accuracy score that was achieved was : **0.9040**
The time that was needed to be achieved was: **77.82** seconds
The classification report is in Table 3

Table 3: Classification Report

	Precision	Recall	F1-Score	Support
airplane	0.91	0.90	0.90	1000
automobile	0.90	0.91	0.90	1000
Accuracy			0.90	2000
Macro Avg	0.90	0.90	0.90	2000
Weighted Avg	0.90	0.90	0.90	2000

The confusion matrix is in Figure 3
We can see similar results to the polynomial kernel. The difference lies in the airplane images, 20 of which it classified as automobiles in comparison with the polynomial kernel

4.1.4 Sigmoid kernel

On the sigmoid kernel, the accuracy score that was achieved was: **0.6925**
The time that was needed to be achieved was: **62.09** seconds
The classification report is in Table 4
The confusion matrix is in Figure 4
We conclude that the sigmoid kernel is not good for this type of classification.

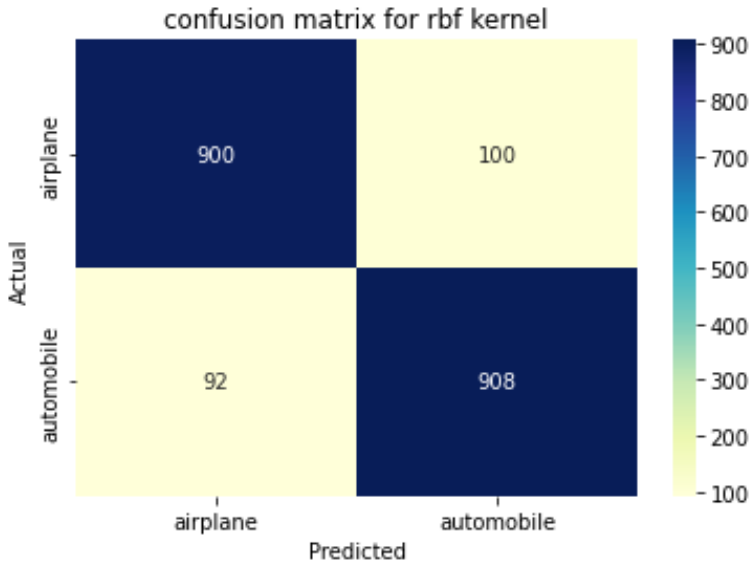


Figure 3: Confusion matrix for rbf kernel

Table 4: Classification Report of Sigmoid kernel

	Precision	Recall	F1-Score	Support
airplane	0.69	0.71	0.70	1000
automobile	0.70	0.68	0.69	1000
Accuracy			0.69	2000
Macro Avg	0.69	0.69	0.69	2000
Weighted Avg	0.69	0.69	0.69	2000

4.1.5 SVM from scratch

In order to run multiple classification tests we used different values for learning rate and lambda parameter.

```
param_grid = {
    'lambda_param': [0.01, 0.1, 1, 10, 100],
    'learning_rate': [0.1, 0.01, 0.001]
}
```

The loop code is added below

```
run = []
iteration = 0

for lambda_param in param_grid['lambda_param']:
    for learning_rate in param_grid['learning_rate']:

        run.append(neptune.init_run(
            project="jason-k/svm-neural",
            name="logging-to-multipe-runs",
            api_token="eyJhcGlfYWRkcmVzcyI6ImhOdHBzOi8vYXBwLm5lcHR1bmUuYWkiLCJhcGlfdXJsIjoiaHROcHM6Ly9hcHAubmVwdH
        ))  # your credentials

        svm_self_made = SVM_from_scratch(learning_rate=learning_rate,
             lambda_param=lambda_param , n_iters= n_iters , iteration = iteration)
        # Train the model and evaluate performance
        # Record the performance metrics for this combination
```

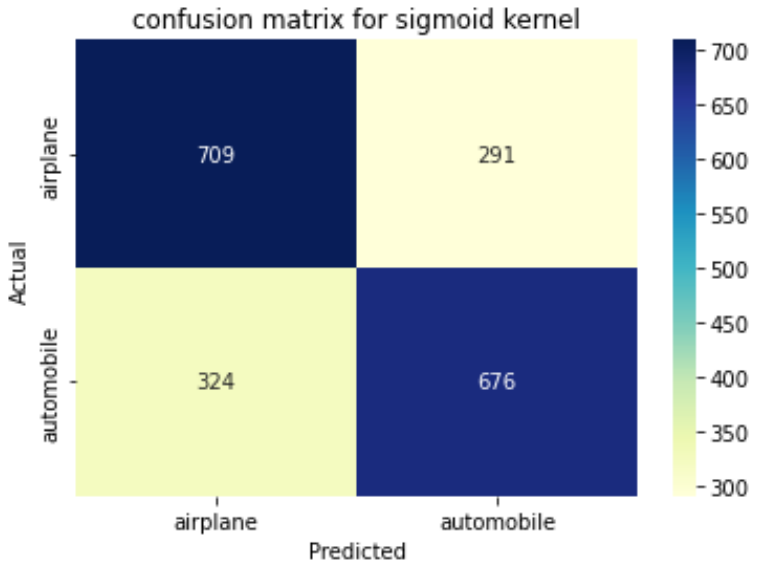


Figure 4: Confusion matrix for sigmoid kernel

```
start_time = time.perf_counter()
svm_self_made.fit(X_train_selected, y_train_selected_numeric)
end_time = time.perf_counter()


predictions = svm_self_made.predict(X_test_selected)


# compute and print accuracy score
print('Model accuracy score for learning rate:', learning_rate ,'and lamda param P:',
    lambda_param ,'is ={0:0.4f}'. format(accuracy_score(y_test_selected_numeric,
    predictions)))
print(classification_report(y_test_selected_numeric, predictions))
print(end_time-start_time)
# Log metrics
accuracy = accuracy_score(y_test_selected_numeric, predictions)
run[iteration]["params/accuracy"].append(accuracy)
#model_version["accuracy"] = accuracy



cm = confusion_matrix(y_test_selected_numeric, predictions)
class_names_selected = ['airplane', 'automobile']

heatmap_1 = sns.heatmap(
     cm, annot=True, fmt='d', cmap='YlGnBu', xticklabels=class_names_selected,
         yticklabels=class_names_selected)
heatmap_1.set_title("confusion matrix for SVM from scratch with {lambda_param} and
    {learning_rate}")
plt.xlabel('Predicted')
plt.ylabel('Actual')

plt.show()

run[iteration].stop()
iteration += 1
```

Furthermore, we used the Neptune AI to create graphs for the accuracy the the iterations. Below we list some of the results for reference.

Note: only the paragraph 4.1.5.1 has n iterations = 200. All the others are 300 so we do not mention it

---

4.1.5.1 learning_rate = 0.01 lambda_param = 0.02 n_iters = 200

We have an accuracy score of : **0.7730**
The elapsed time was: **38.75** seconds
The classification report can be seen in Table 5

Table 5: Classification Report for self made SVM

	Precision	Recall	F1-Score	Support
airplane	0.73	0.87	0.79	1000
automobile	0.84	0.68	0.75	1000
Accuracy			0.77	2000
Macro Avg	0.78	0.77	0.77	2000
Weighted Avg	0.78	0.77	0.77	2000

4.1.5.2 learning rate 0.1 and lambda param 0.01

We have an accuracy score of : **0.7985**
The elapsed time was: **58.89** seconds
The classification report can be seen in Table 6

Table 6: Classification Report for SVM with Learning Rate 0.1 and Lambda Parameter 0.01

	Precision	Recall	F1-Score	Support
airplane	0.80	0.80	0.80	1000
automobile	0.80	0.80	0.80	1000
Accuracy			0.80	2000
Macro Avg	0.80	0.80	0.80	2000
Weighted Avg	0.80	0.80	0.80	2000

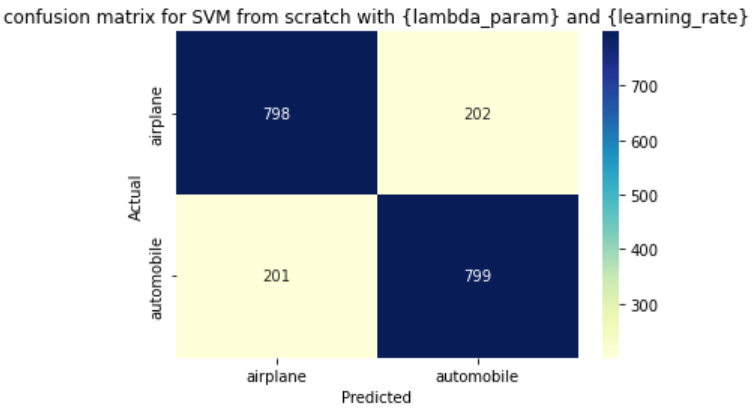


Figure 5: Enter Caption

Graph of accuracy from Neptune in 6
We see overfitting

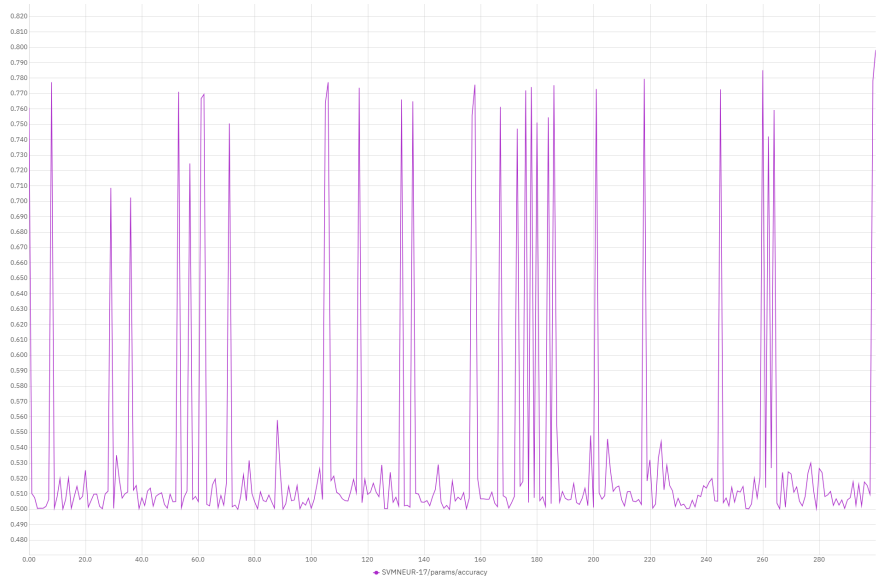


Figure 6: learning rate 0.1 and lambda param 0.01

4.1.5.3 Model accuracy score for learning rate: 0.01 and lamda param: 0.01 We have an accuracy score of : **0.7140**
The elapsed time was: **54.77** seconds
The classification report can be seen in Table 7

Table 7: Classification Report for SVM with Learning Rate 0.01 and Lambda Parameter 0.01

	Precision	Recall	F1-Score	Support
-1	0.89	0.49	0.63	1000
1	0.65	0.94	0.77	1000
Accuracy			0.71	2000
Macro Avg	0.77	0.71	0.70	2000
Weighted Avg	0.77	0.71	0.70	2000

Graph of accuracy from Neptune in 8

4.1.5.4 learning rate: 0.001 and lamda param: 0.01 We have an accuracy score of : **0.8250**
The elapsed time was: **52.6** seconds
The classification report can be seen in Table 8

Table 8: Classification Report for SVM with Learning Rate 0.001 and Lambda Parameter 0.01

	Precision	Recall	F1-Score	Support
-1	0.82	0.84	0.83	1000
1	0.83	0.81	0.82	1000
Accuracy			0.82	2000
Macro Avg	0.83	0.82	0.82	2000
Weighted Avg	0.83	0.82	0.82	2000

Graph of accuracy from Neptune in 10

4.1.5.5 learning rate: 0.001 and lamda param: 0.1 We have an accuracy score of : **0.8185**
The elapsed time was: **55.8** seconds
The classification report can be seen in Table 9
Graph of accuracy from Neptune in 12

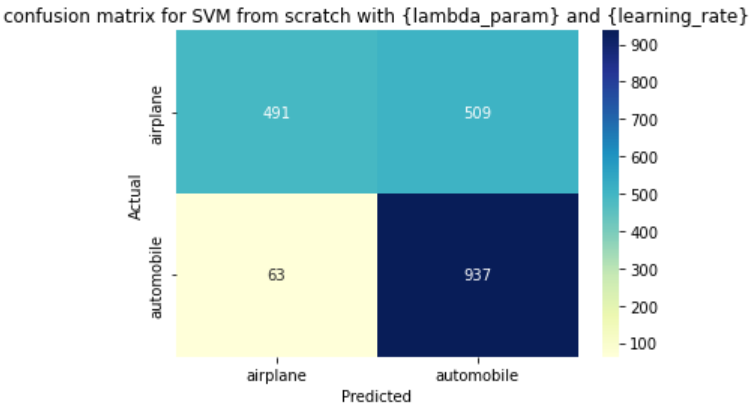


Figure 7: svm_lr0.01_l0.01

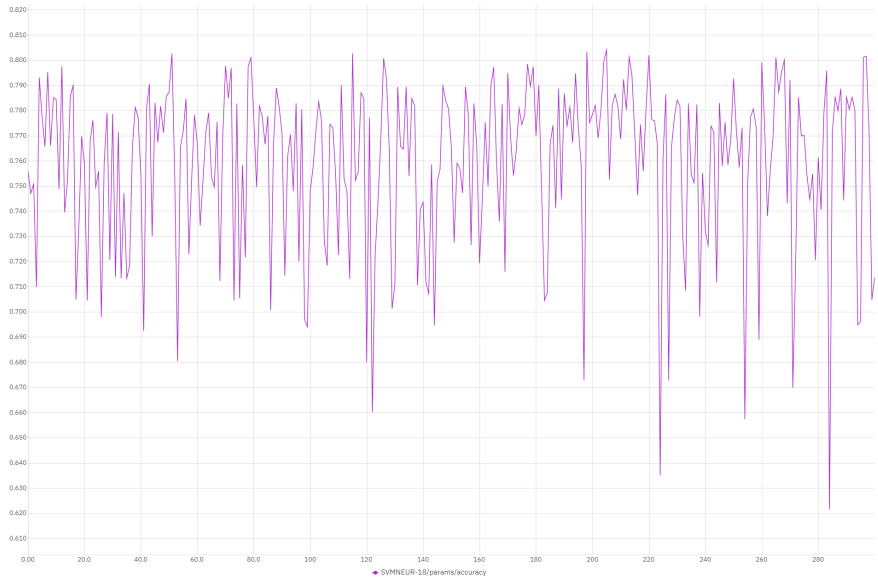


Figure 8: svm_lr0.01_l0.01

4.1.5.6 Rest runs All the other runs had either accuracy = 0.5 or close to 0.65 so they won't be mentioned here.

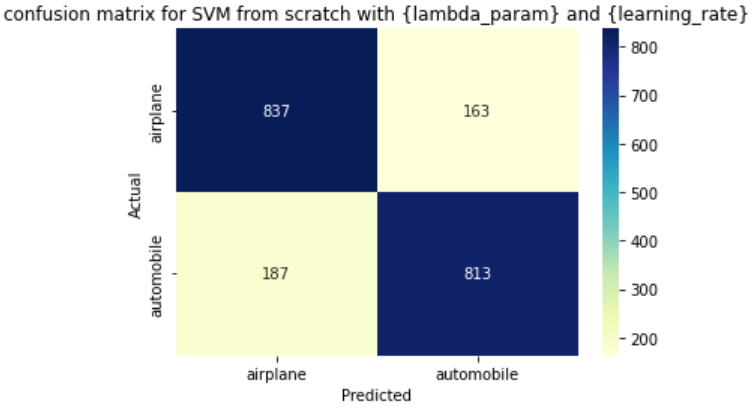


Figure 9: svm_lr0.001_l0.01

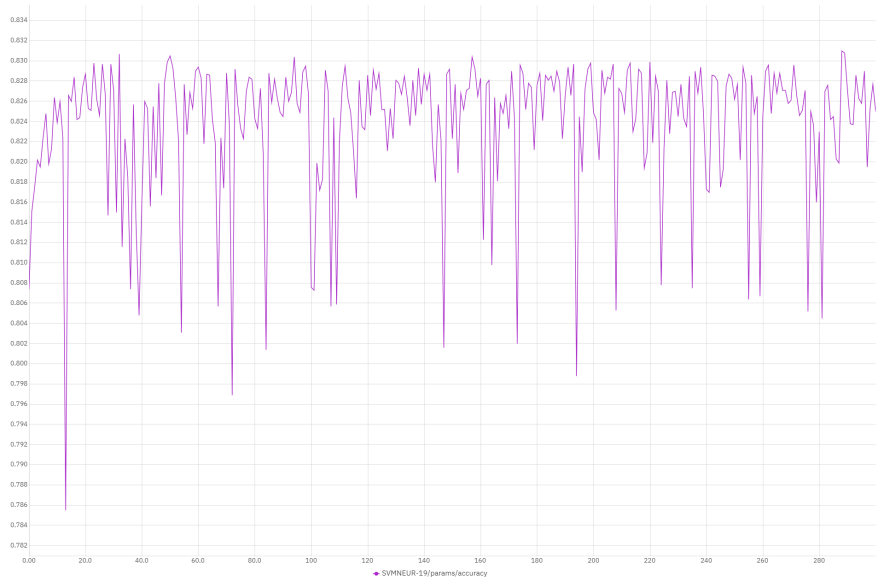


Figure 10: svm_lr0.001_l0.01

Table 9: Classification Report for SVM with Learning Rate 0.001 and Lambda Parameter 0.1

	Precision	Recall	F1-Score	Support
-1	0.83	0.80	0.81	1000
1	0.80	0.84	0.82	1000
Accuracy			0.82	2000
Macro Avg	0.82	0.82	0.82	2000
Weighted Avg	0.82	0.82	0.82	2000

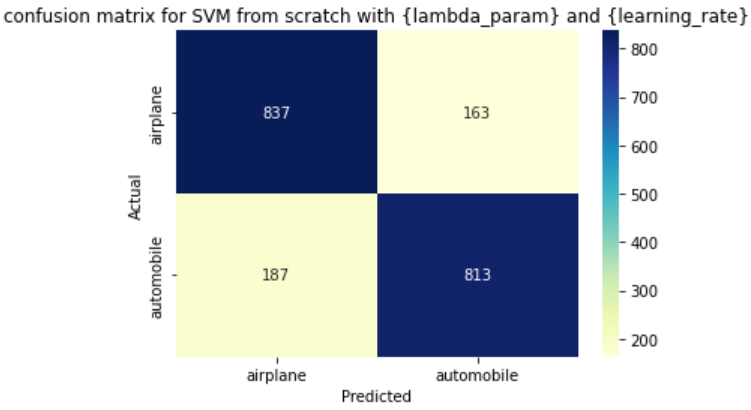


Figure 11: svm_lr0.001_l0.1

Figure 12: svm_lr0.001_l0.1

4.2 multiple batched size classification

We will use the previous kernels for the batched multi-classification with the exemption of the custom-made SVM which needs more modifications in order to process this data

4.2.1 Polynomial Kernel

We have an accuracy score of : **0.4240**
The elapsed time was: **203.72** seconds
The classification report can be seen in Table 10
The matrix can be seen in Figure 13

4.2.2 GridSearch

In order to run the GridSearch function we simplified the version mentioned in section 2.6. We will use the below alternations of variables and check the results.

```
parameters = [ {'C':[1, 10], 'kernel':['linear']},
               {'C':[1, 10], 'kernel':['rbf'], 'gamma':[0.1, 0.5, 0.9]},
               {'C':[1, 10], 'kernel':['poly'], 'degree': [2,4] ,'gamma':[0.01, 0.05]}
               ]
```

Table 10: Classification Report for Multiclass SVM

	Precision	Recall	F1-Score	Support
0	0.46	0.46	0.46	196
1	0.54	0.58	0.56	198
2	0.26	0.30	0.28	195
3	0.31	0.32	0.31	199
4	0.31	0.35	0.33	198
5	0.34	0.26	0.30	185
6	0.46	0.50	0.48	216
7	0.51	0.44	0.47	193
8	0.53	0.59	0.56	217
9	0.54	0.41	0.46	203
Accuracy			0.42	2000
Macro Avg	0.43	0.42	0.42	2000
Weighted Avg	0.43	0.42	0.42	2000

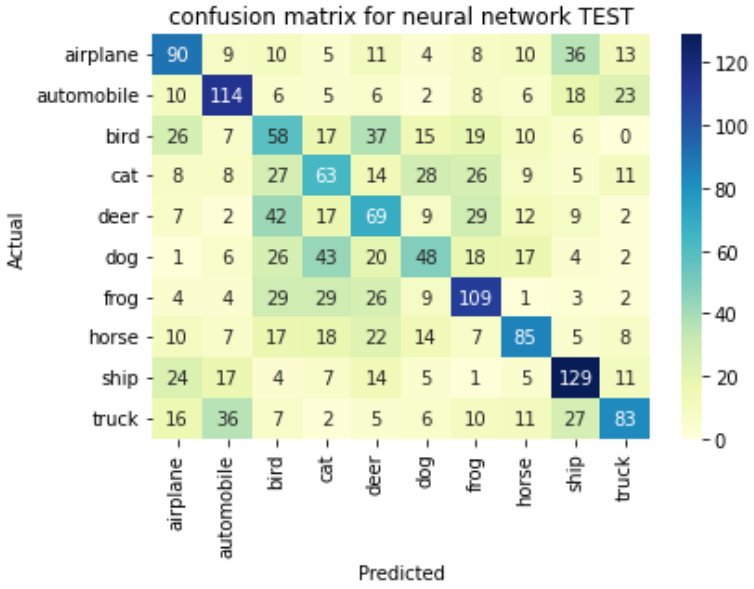


Figure 13: multiclass_svm

The results are listed in 11

The parameters that gave the best results are **'C': 1, 'degree': 2, 'gamma': 0.01, 'kernel': 'poly'**

Score on the Test set : **0.4220**

Table 11: Grid Search Results for SVM Hyperparameters

C	Kernel	Gamma	Degree	Mean Fit Time	Std Fit Time	Mean Score Time	Std Score Time	Mean Test Score	Std Test Score	Rank
1	Linear	-	-	154.51	19.15	23.45	2.25	0.3108	0.0087	9
10	Linear	-	-	168.69	3.17	22.25	0.35	0.3016	0.0051	10
1	RBF	0.1	-	186.76	1.28	40.70	0.46	0.1374	0.0045	12
1	RBF	0.5	-	186.19	1.28	40.27	0.62	0.1058	0.0002	13
1	RBF	0.9	-	185.57	2.42	40.76	1.10	0.1058	0.0002	13
10	RBF	0.1	-	185.35	1.42	39.68	0.72	0.1481	0.0056	11
10	RBF	0.5	-	185.65	1.68	39.95	1.02	0.1058	0.0002	13
10	RBF	0.9	-	190.07	5.63	41.45	0.84	0.1058	0.0002	13
1	Poly	0.01	2	113.21	2.73	22.75	0.86	0.4194	0.0129	1
1	Poly	0.05	2	128.22	0.78	22.27	0.28	0.4032	0.0072	4
1	Poly	0.01	4	101.56	1.26	20.84	0.22	0.4032	0.0058	4
1	Poly	0.05	4	97.84	2.60	20.64	0.26	0.39	0.0083	7
10	Poly	0.01	2	128.72	1.89	22.15	0.33	0.4035	0.0074	3
10	Poly	0.05	2	127.26	1.28	22.17	0.29	0.4037	0.0069	2
10	Poly	0.01	4	99.09	2.12	20.85	0.30	0.3917	0.0082	6
10	Poly	0.05	4	97.50	2.65	20.89	0.13	0.39	0.0083	7

5 Conlcusions

From the results we can conclude that:

- The SVM with the polynomial kernel had the best accuracy score for 2 class classifications equal to **0.9145**
- For the multi-batched classification the polynomial kernel had a score of **0.4480** and **203.71** seconds. Considerably lower than the 2-class classification and slower.
- The custom made SVM with linear kernel had the best results by using **learning rate=0.001 and lamda param=0.01**. The accuracy score was **0.8250** lower than the polynomial kernel but a bit better than the linear kernel of the sklearn library
- Using the GridSearch function with different kernel and different combinations of values we concluded that the best results was using **the polynomial kernel 2nd degree with gamma= 0.01 and C=1**. The accuracy score on the test set was: **0.4220**

References

[Ass] AssemblyAI. How to implement svm (support vector machine) from scratch with python.