UNIVERSITY OF WEST LONDON
The Career University

Machine Learning Log Book

Name: Aman Adish
Student ID: 21587065
Batch: L6 - Batch 2

Contents

Week – 3 – Unsupervised Learning (Clustering) 4
Collab Links 4
Task 1 5
Task 2 5
Week – 4 – Classification (K-NN and Performance Metrics) 7
Collab Links 8
Task 1 8
Task 2 10
Task 3 16
Week – 5 – Classification (Random Forest) 17
Collab Links 17
Task 1 17
A) Subset of 2 Features 18
B) Subset of 3 Features 21
C) Subset of 4 Features 24
Task 2 28
Week – 6 – Classification (Linear Classifiers) 30
Collab Links 30
Task 1 30
Week – 7 – Classification (Support Vector Machines) 35
Collab Links 35
Task-1 35
c = 0.01 38
c = 0.1 39
c = 1.0 40
Task-2 41
Week – 8 – Classification (Neural Networks) 44

Collab Links 44
Task 1 44
Task 2 45
Week – 9 – Classification (Image Classification) 50
Collab Links 50
Task 1 50
Task 2 51
Week – 10 – Classification (Convolutional Neural Network) 62
Collab Links 62
Task 1 62
Task 2 64
Week – 11 – Supervised Learning (Regression) 88
Collab Link 88
Task1 88
Task 2 90
Task 3 92
Task 4 95

Week – 3 – Unsupervised Learning (Clustering)

Clustering is **a type of Unsupervised learning**.

In Hierarchical Clustering which is **a "bottom-up" approach** each observation starts in its own cluster, and **pairs of clusters are merged** as one moves up the hierarchy. (**Agglomerative Strategy**)

Dendrogram shows the merging of clusters during hierarchical clustering.

- Using **Linkages,** the data points are clustered
- Shows the **hierarchy of clusters** and helps us choose the right number of clusters.

Ward's Method

It **minimizes the variance within clusters** as they are merged.

[1 1 1 1 1 0 0 0 0 0]

- The first 5 points belong to Cluster 1
- The last 5 points belong to Cluster 0

K- means Clustering

How does the clustering works?

- Pick K random points as cluster centers called **centroids**
- Assign each input to **nearest cluster** by calculating its distance to each centroid
- Find **new cluster** center by taking the **average of the assigned points**
- Repeat Step 2 and 3 until **none of the cluster assignments change**

Collab Links

Part 1

https://colab.research.google.com/drive/1VcUjbhEmEdVr-Ox-eekVfl1IsfTTyzv9?usp=sharing

Part 2

https://colab.research.google.com/drive/1G55eQZzz0yHju8zAMKRT_lLmGadRPLOY?usp=sharing

Task 1

Describe the issues associated with K-means algorithm, and explain the mitigating options.

Ans: Prone to be more **favorable towards to starting points**, the **clusters formed depend on where the centroids are initially placed**. The centroids are chosen randomly, unthought of starting points can lead to bad results.

Using methods like K-means++ or try different starting points multiple times and choose the best outcome. This can be unnecessarily time consuming. Another issue is that, K-means requires the **user to specify the number of clusters in advance**, which might not be known, using the **elbow method** can be used to estimate the optimal number of clusters.

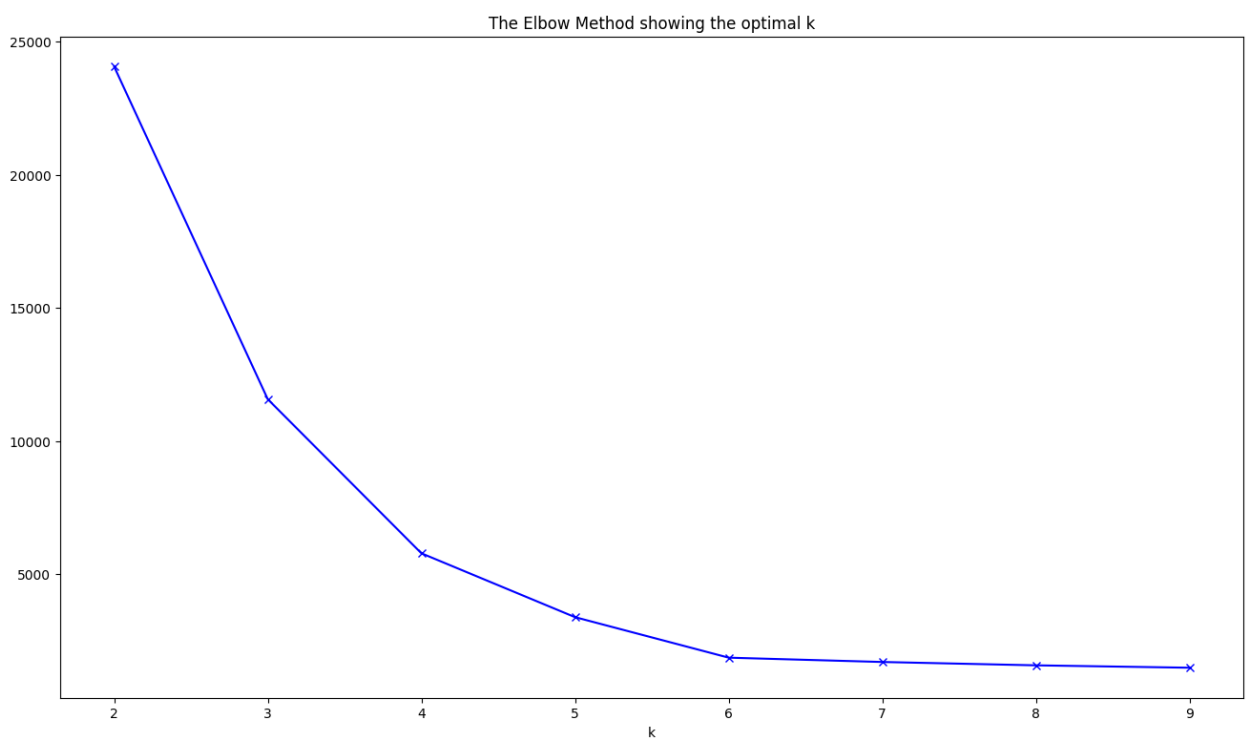
Task 2

The dataset we are going to use has 1000 entries with unknown number of clusters (download data_Kclusters.txt from Blackboard). Use the **Elbow Method to find the optimum value for K** and show the results for clustering using the optimum k. You can use the following lines to plot the elbow.

```
[ ] WCSS = []
    K = range(2,10)
    for this_k in K:
        kmeanModel = KMeans(n_clusters = this_k)
        kmeanModel.fit(X)
        WCSS.append(kmeanModel.inertia_)

    # Plot the elbow
    plt.figure()
    plt.plot(K, WCSS, 'bx-')
    plt.xlabel('k')
    plt.ylabel('WCSS')
    plt.title('The Elbow Method showing the optimal k')
    plt.show()
```

Ans:

```
[ ]  # Initializing KMeans
     K_elbow = 5
     kmeans = KMeans(n_clusters = K_elbow)
     # Fitting with inputs
     kmeans = kmeans.fit(X)
     # Predicting the clusters
     labels = kmeans.predict(X)
     # Getting the cluster centers
     C = kmeans.cluster_centers_

     plt.figure()
     plt.scatter(X[:, 0], X[:, 1], c=labels)
     #plt.scatter(X[:, 0], X[:, 1], c=labels)
     plt.scatter(C[:, 0], C[:, 1],  marker='*', c='#050505', s=1000)
     plt.show()
```



Using the Elbow method, the optimal value **of K is found to be 6.**

Week – 4 – Classification (K-NN and Performance Metrics)

The output of a KNN Classification is a **class membership**, predicts a class.

K-NN is also a **lazy algorithm**. What this means is that it **does not use the training data points to do any generalization**. Which means that K-NN keeps all the training data. To be more exact, **all (or most) the training data is needed during the testing phase**.

The **Confusion matrix** is used for Classification problem where the output can be of **two or more types of classes**.

- have cancer
- don't have cancer

The Confusion Matrix is a 2D Table with its sets of classes.

- **True Positives** (TP) - Actual (1) Predicted (1)
- **True Negatives** (TN) - Actual (0) Predicted (0)
- **False Positives** (FP) - Actual (0) Predicted (1)
- **False Negatives** (FN) - Actual (1) Predicted (0)

Accuracy = $\frac{TP+TN}{\mathrm{TP + TN + FP + FN}}$

Accuracy in classification problems is the **number of correct predictions made by the model over all kinds predictions made**.

Precision = TP / TP + FP

Precision in classification problems is a **measure of how correctly the algorithm predicted**.

Recall = TP / TP + FN

Recall is a metric that measures **how often a machine learning model correctly identifies positive instances** (true positives) from all the actual positive samples in the dataset

Specificity = TN / TN + FP

Sensitivity measures how well a machine learning model **can detect positive instances**.

Collab Links

https://colab.research.google.com/drive/174LV4Z_Oj_mnAXBAWLzCDamQgQguZMdF?usp=sharing

Task 1

Let us assume we developed a machine learning model to predict which patients on dialysis will be admitted to the hospital in the next week. Given 100 patients, we have the following break-down:

• True Positives: people that are hospitalized that you predict will be hospitalized

• True Negatives: people that are NOT hospitalized that you predict will NOT be hospitalized

• False Positives: people that are NOT hospitalized that you predict will be hospitalized

• False Negatives: people that are hospitalized that you predict will NOT be hospitalized

Calculate the performance metrics in the table below:

Confusion Matrix		Actual	
		Hospitalized	Not Hospitalized
Predicted	Hospitalized	33	10
	Not Hospitalized	17	40

Recall (Sensitivity)	
Specificity	
Accuracy	
Precision	

Ans. Using the formulas for Accuracy, Precision, Recall and Specificity, we can easily find out the:

TP	33
TN	40
FP	10
FN	17

- **Accuracy**

Accuracy = $\frac{TP+TN}{\mathrm{TP+TN+FP+FN}}$ = $\frac{(33+40)}{(33+40+10+17)}$ = **0.73**

- **Precision**

Precision = $\frac{TP}{\mathrm{TP+FP}}$ = $\frac{33}{(33+10)}$ = **0.767**

- **Recall (Sensitivity)**

True Positives (TP) = 33	False Negatives (FN) = 17

Recall = $\frac{TP}{\mathrm{TP+FN}}$ = $\frac{33}{(33+17)}$ = **0.66**

- **Specificity**

True Negatives (TN) = 40	False Positives (FP) = 10

Specificity = $\frac{TN}{\mathrm{TN+FP}}$ = $\frac{40}{(40+10)}$ = **0.80**

Performance Metrics	
Recall (Sensitivity)	0.66
Specificity	0.80
Accuracy	0.73
Precision	0.767

Task 2

Try K-NN classifier for dataset provided in for different numbers of neighbors (K = 3 and K = 7). Which K provides better results? Justify your answer by using AUC of ROC curve. Also provide the relevant performance metrics such sensitivity, etc.

Ans:

When K=3

```
[5] from sklearn.neighbors import KNeighborsClassifier
    model = KNeighborsClassifier(n_neighbors=3)
    model.fit(trainX, trainy)
```

```
KNeighborsClassifier
KNeighborsClassifier(n_neighbors=3)
```

```
[6] from sklearn.metrics import confusion_matrix
    cnf_matrix = confusion_matrix(testy, model.predict(testX))

    TP = cnf_matrix[0][0]
    FP = cnf_matrix[0][1]
    FN = cnf_matrix[1][0]
    TN = cnf_matrix[1][1]
    print('TP = ', TP)
    print('FP = ', FP)
    print('FN = ', FN)
    print('TN = ', TN)

    print('\nPerformance Metrics')
    #Sensitivity, hit rate, recall, or true positive rate
    print('TPR = Sensitivity = recal = '+'{:.2f}'.format(TP/(TP+FN)))
    # Specificity or true negative rate
    print('TNR = Specificity = '+'{:.2f}'.format(TN/(TN+FP)))
    # Precision or positive predictive value
    print('PPV = Precision = '+'{:.2f}'.format(TP/(TP+FP)))
    # Overall accuracy
    print('ACC = Accuracy = '+'{:.2f}'.format((TP+TN)/(TP+FP+FN+TN)))
```

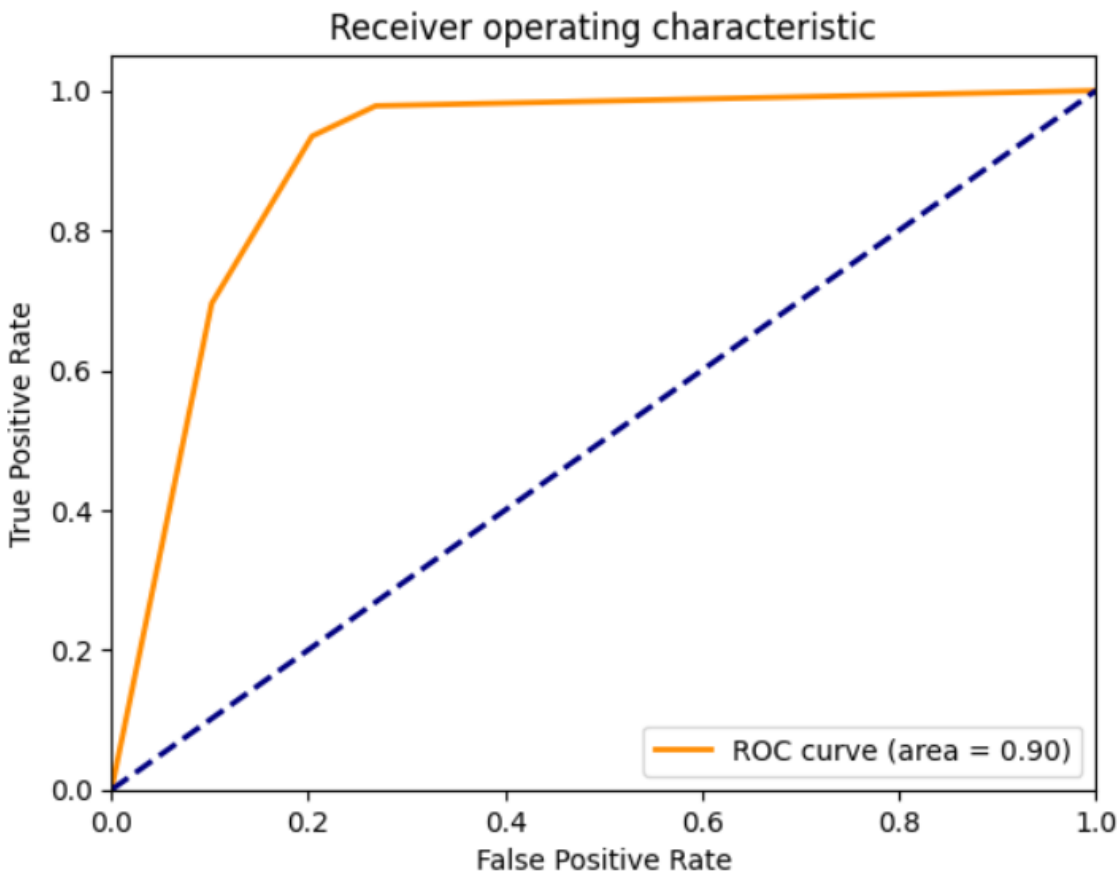
[6]
```
TP =  86
FP =  22
FN =  6
TN =  86

Performance Metrics
TPR = Sensitivity = recal = 0.93
TNR = Specificity = 0.80
PPV = Precision = 0.80
ACC = Accuracy = 0.86
```

[7]
```
# print('\nPerformance Metrics using sklearn')
# from sklearn.metrics import accuracy_score
# print('accuracy = '+'{:.2f}'.format(accuracy_score(testy, model.predict(testX))))
# print('accuracy = '+'{:.2f}'.format(model.score(testX, testy)))

# from sklearn.metrics import precision_score
# print('precision = '+'{:.2f}'.format(precision_score(testy, model.predict(testX), pos_label=1)))

# from sklearn.metrics import recall_score
# print('recall = '+'{:.2f}'.format(recall_score(testy, model.predict(testX), pos_label=1)))

from sklearn.metrics import f1_score
print('f1_score = '+'{:.2f}'.format(f1_score(testy, model.predict(testX), pos_label=1)))

# from sklearn.metrics import classification_report
# print(classification_report(testy, model.predict(testX), target_names=['Negative', 'Positive']))
```

```
f1_score = 0.86
```

[10]
```
from sklearn.metrics import roc_curve
from sklearn.metrics import roc_auc_score
## predict probabilities
probs = model.predict_proba(testX)
# keep probabilities for the positive outcome only
probs = probs[:, 1]
# calculate roc curve
fpr, tpr, thresholds = roc_curve(testy, probs)
# calculate AUC
auc = roc_auc_score(testy, probs)
print('\nArea Under The Curve: AUC = %.3f\n' % auc)


plt.figure()
lw = 2
plt.plot(fpr, tpr, color='darkorange', lw=lw, label='ROC curve (area = %0.2f)' % auc)
plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic')
plt.legend(loc="lower right")
plt.show()
```

```

Area Under The Curve: AUC = 0.904
```



K = 3

f1_score = 0.86

AUC = 0.904

When K=7

```
[15] from sklearn.neighbors import KNeighborsClassifier
     model = KNeighborsClassifier(n_neighbors=7)
     model.fit(trainX, trainy)
```

```
KNeighborsClassifier
KNeighborsClassifier(n_neighbors=7)
```

```
[16] from sklearn.metrics import confusion_matrix
     cnf_matrix = confusion_matrix(testy, model.predict(testX))

     TP = cnf_matrix[0][0]
     FP = cnf_matrix[0][1]
     FN = cnf_matrix[1][0]
     TN = cnf_matrix[1][1]
     print('TP = ', TP)
     print('FP = ', FP)
     print('FN = ', FN)
     print('TN = ', TN)

     print('\nPerformance Metrics')
     #Sensitivity, hit rate, recall, or true positive rate
     print('TPR = Sensitivity = recal = '+'{:.2f}'.format(TP/(TP+FN)))
     # Specificity or true negative rate
     print('TNR = Specificity = '+'{:.2f}'.format(TN/(TN+FP)))
     # Precision or positive predictive value
     print('PPV = Precision = '+'{:.2f}'.format(TP/(TP+FP)))
     # Overall accuracy
     print('ACC = Accuracy = '+'{:.2f}'.format((TP+TN)/(TP+FP+FN+TN)))
```

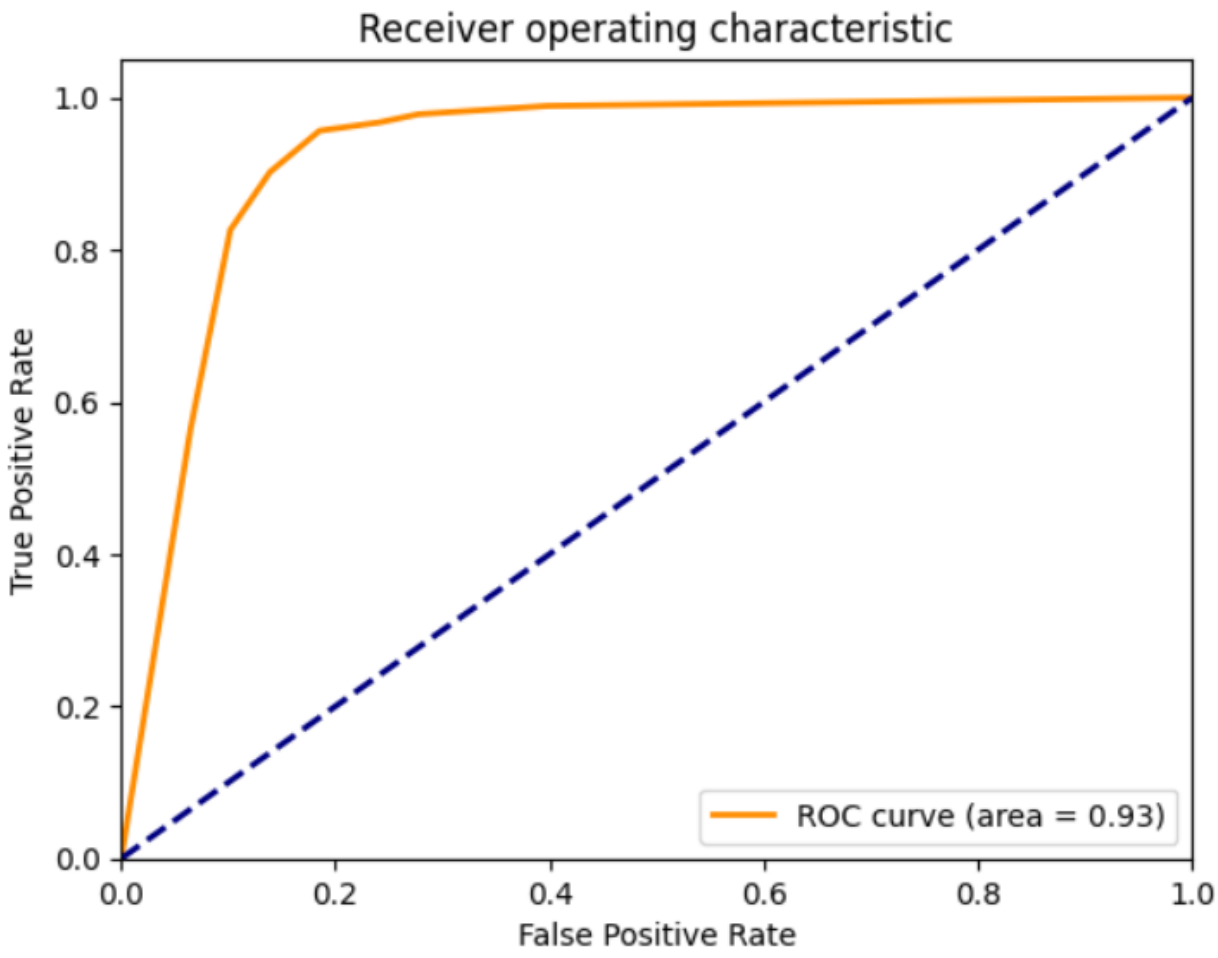
[16]
```
TP =  88
FP =  20
FN =  4
TN =  88

Performance Metrics
TPR = Sensitivity = recal = 0.96
TNR = Specificity = 0.81
PPV = Precision = 0.81
ACC = Accuracy = 0.88
```

[17]
```
# print('\nPerformance Metrics using sklearn')
# from sklearn.metrics import accuracy_score
# print('accuracy = '+'{:.2f}'.format(accuracy_score(testy, model.predict(testX))))
# print('accuracy = '+'{:.2f}'.format(model.score(testX, testy)))

# from sklearn.metrics import precision_score
# print('precision = '+'{:.2f}'.format(precision_score(testy, model.predict(testX), pos_label=1)))

# from sklearn.metrics import recall_score
# print('recall = '+'{:.2f}'.format(recall_score(testy, model.predict(testX), pos_label=1)))

from sklearn.metrics import f1_score
print('f1_score = '+'{:.2f}'.format(f1_score(testy, model.predict(testX), pos_label=1)))

# from sklearn.metrics import classification_report
# print(classification_report(testy, model.predict(testX), target_names=['Negative', 'Positive']))
```

```
f1_score = 0.88
```

[30]
```
from sklearn.metrics import roc_curve
from sklearn.metrics import roc_auc_score
## predict probabilities
probs = model.predict_proba(testX)
# keep probabilities for the positive outcome only
probs = probs[:, 1]
# calculate roc curve
fpr, tpr, thresholds = roc_curve(testy, probs)
# calculate AUC
auc = roc_auc_score(testy, probs)
print('\nArea Under The Curve: AUC = %.3f\n' % auc)


plt.figure()
lw = 2
plt.plot(fpr, tpr, color='darkorange', lw=lw, label='ROC curve (area = %0.2f)' % auc)
plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic')
plt.legend(loc="lower right")
plt.show()
```

```
Area Under The Curve: AUC = 0.926
```



K = 7

f1_score = 0.88

AUC = 0.926

Conclusion

Since **AUC K=7 > AUC K=3**, **K=7 provides better classification performance** in terms of overall discrimination between classes.

Task 3

The script also retunes a value called ‘f1_score’. Do some research to understand what this measure is. Repeat task-2 and this time justify your answer using f1_score. Do you arrive at the same conclusion?

Ans: The **f1-score is the mean of precision and recall,** making it particularly useful when dealing with imbalanced datasets or when both false positives and false negatives are significant.

$$\mathrm{F1} = \frac{2\ X\ Precision\ X\ Recall}{\mathrm{Precision+Recall}}$$

The f1-score **ranges from 0 to 1**, where 1 indicates perfect precision and recall and 0 indicates the worst possible performance, where either precision or recall is zero. The F1-score is particularly useful in scenarios where there is an uneven class distribution, as it provides a single metric that balances the trade-off between precision and recall.

Conclusion

f1-score of K’s = 7 is higher than f1-score of K = 3, suggesting that K = 7 provides better overall classification performance. So, we can arrive at same the conclusion that both the **AUC of ROC curve and f1-score indicate that K=7 outperforms K=3** for this dataset.

Week – 5 – Classification (Random Forest)

Random Forest algorithm is a powerful tree learning technique in Machine Learning. It works by **creating a number of Decision Trees during the training phase**. Each tree is constructed using a random subset of the data set to measure a random subset of features in each partition. This **randomness introduces variability** among individual trees, **reducing the risk of overfitting** and improving overall prediction performance.

Most Powerful Parts of Random Forest - **View Feature Importance**

We get a score telling us how important each feature was in classifying. This is one of the most powerful parts of random forests.

Collab Links

https://colab.research.google.com/drive/13VeLcvLESN6GNdezqJUZ039QHt5XPrFz?usp=sharing

Task 1

Try Random Forest Classifier for the Iris dataset example, and when creating decision trees from the bootstrapped dataset, use different number of features:

- a subset of 2 features
- a subset of 3 features
- a subset of 4 features

And compare the accuracy of your Random Forest Classifier. Which one is better? Discuss your observations and justify your answer. What is the largest number of features you may have?

Ans: Loading the Iris Dataset

```
# Load the library with the iris dataset
from sklearn.datasets import load_iris
# Create an object called iris with the iris data
iris = load_iris()

# Create a dataframe with the four feature variables
df = pd.DataFrame(iris.data, columns=iris.feature_names)
# View the top 5 rows
df.head()
```

A) Subset of 2 Features

Training The Random Forest Classifier

```
[ ] # Load scikit's random forest classifier library
    from sklearn.ensemble import RandomForestClassifier

    # Create a random forest Classifier. By convention, clf means 'Classifier'
    # We can specify the number of features using max_features
    clf = RandomForestClassifier(max_features=2, random_state=0)

    # Train the Classifier to take the training features and learn how they relate
    # to the training y (the species)
    clf.fit(train[features], y)
```

```
RandomForestClassifier
RandomForestClassifier(max_features=2, random_state=0)
```

Applying Classifier To Test Data

We leave out a portion of the data so that we have a set of data to test the accuracy of the model

```
[ ] # Apply the Classifier we trained to the test data (which, remember, it has never seen before)
    y_pred = clf.predict(test[features])
    y_pred
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 2, 2, 1, 1, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

This is what species the model predicted each plant is based on the the sepal length, sepal width, petal length, and petal width.

```
[ ] # View the predicted probabilities of the first 10 observations
    clf.predict_proba(test[features])[0:25]
```

```
array([[1.  , 0.  , 0.  ],
       [1.  , 0.  , 0.  ],
       [1.  , 0.  , 0.  ],
       [1.  , 0.  , 0.  ],
       [1.  , 0.  , 0.  ],
       [0.95, 0.05, 0.  ],
       [1.  , 0.  , 0.  ],
       [0.99, 0.01, 0.  ],
       [1.  , 0.  , 0.  ],
       [1.  , 0.  , 0.  ],
       [1.  , 0.  , 0.  ],
       [0.99, 0.01, 0.  ],
       [1.  , 0.  , 0.  ],
       [0.  , 0.67, 0.33],
       [0.  , 1.  , 0.  ],
       [0.  , 0.82, 0.18],
       [0.  , 0.03, 0.97],
       [0.  , 0.42, 0.58],
       [0.  , 0.99, 0.01],
       [0.  , 0.96, 0.04],
       [0.  , 0.  , 1.  ],
       [0.  , 0.  , 1.  ],
       [0.  , 0.  , 1.  ],
       [0.  , 0.01, 0.99],
       [0.  , 0.  , 1.  ]])
```

The classifier gives a 90% probability the plant belongs to the first class and a 10% to the plant belongs to the second class Because 90 is greater than 10, the classifier predicts the plant is the first class.

Evaluating The Classifier

Now that we have predicted the species of all plants in the test data, we can compare our predicted species with the that plant's actual species.

```
[ ] # Create actual english names for the plants for each predicted plant class
    preds = iris.target_names[clf.predict(test[features])]
```

```
[ ] # View the PREDICTED species for the first five observations
    preds[0:5]
```

```
array(['setosa', 'setosa', 'setosa', 'setosa', 'setosa'], dtype='<U10')
```

```
[ ] # View the ACTUAL species for the first five observations
    test['species'].head()
```

	species
7	setosa
8	setosa
10	setosa
13	setosa
17	setosa

dtype: category

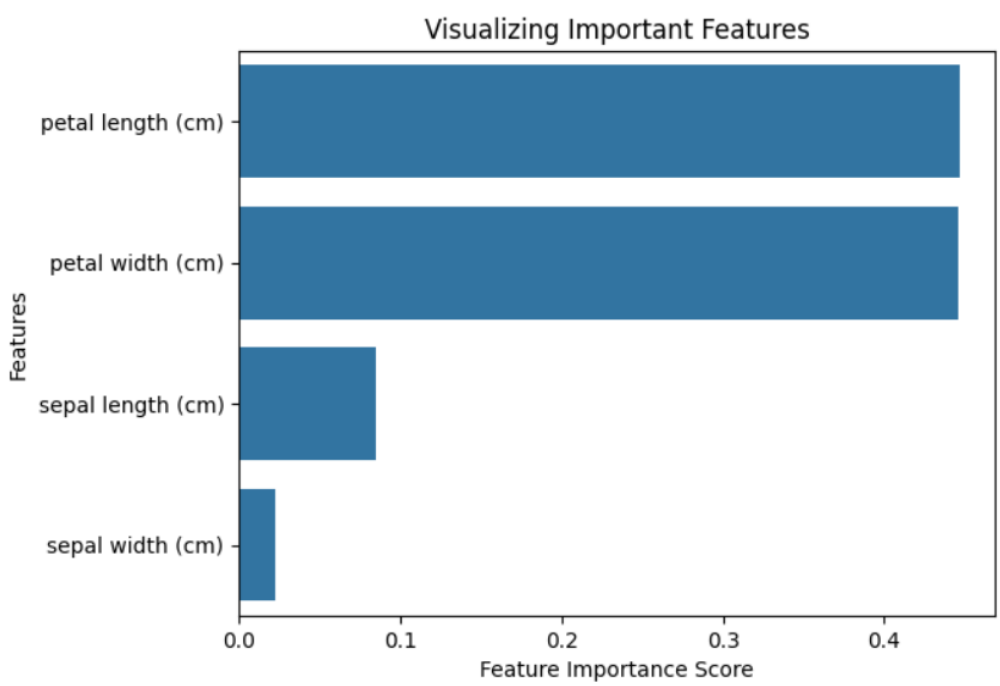
```
[ ] # View a list of the features and their importance scores
    #list(zip(train[features], clf.feature_importances_))
    feature_imp = pd.Series(clf.feature_importances_,index=iris.feature_names).sort_values(ascending=False)
    feature_imp
```

	0
petal length (cm)	0.446485
petal width (cm)	0.446313
sepal length (cm)	0.084740
sepal width (cm)	0.022461

dtype: float64

Visualisation

```
[ ] import matplotlib.pyplot as plt
    import seaborn as sns

    # Creating a bar plot
    sns.barplot(x=feature_imp, y=feature_imp.index)

    # Add labels to your graph
    plt.xlabel('Feature Importance Score')
    plt.ylabel('Features')
    plt.title("Visualizing Important Features")
    plt.show()
```



```
[ ] y_pred = clf.predict(test[features])
    y_pred

    y_test = pd.factorize(test['species'])[0]
    y_test

    #Import scikit-learn metrics module for accuracy calculation
    from sklearn import metrics

    # Model Accuracy, how often is the classifier correct?
    print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.9375
```

B) Subset of 3 Features

Training The Random Forest Classifier

```
[9] # Load scikit's random forest classifier library
    from sklearn.ensemble import RandomForestClassifier

    # Create a random forest Classifier. By convention, clf means 'Classifier'
    # We can specify the number of features using max_features
    clf = RandomForestClassifier(max_features=3, random_state=0)

    # Train the Classifier to take the training features and learn how they relate
    # to the training y (the species)
    clf.fit(train[features], y)
```

```
RandomForestClassifier
RandomForestClassifier(max_features=3, random_state=0)
```

Applying Classifier To Test Data

We leave out a portion of the data so that we have a set of data to test the accuracy of the model

```
[10] # Apply the Classifier we trained to the test data (which, remember, it has never seen before)
     y_pred = clf.predict(test[features])
     y_pred
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 2, 1, 1, 1, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

```
[11] # View the predicted probabilities of the first 10 observations
     clf.predict_proba(test[features])[0:25]
```

```
array([[1.  , 0.  , 0.  ],
       [1.  , 0.  , 0.  ],
       [1.  , 0.  , 0.  ],
       [1.  , 0.  , 0.  ],
       [1.  , 0.  , 0.  ],
       [1.  , 0.  , 0.  ],
       [1.  , 0.  , 0.  ],
       [1.  , 0.  , 0.  ],
       [1.  , 0.  , 0.  ],
       [1.  , 0.  , 0.  ],
       [1.  , 0.  , 0.  ],
       [1.  , 0.  , 0.  ],
       [1.  , 0.  , 0.  ],
       [0.  , 0.69, 0.31],
       [0.  , 1.  , 0.  ],
       [0.  , 0.9 , 0.1 ],
       [0.  , 0.03, 0.97],
       [0.  , 0.5 , 0.5 ],
       [0.  , 1.  , 0.  ],
       [0.  , 0.99, 0.01],
       [0.  , 0.  , 1.  ],
       [0.  , 0.  , 1.  ],
       [0.  , 0.  , 1.  ],
       [0.  , 0.02, 0.98],
       [0.  , 0.  , 1.  ]])
```

The classifier gives a 90% probability the plant belongs to the first class and a 10% the plant belongs to the second class

Because 90 is greater than 10, the classifier predicts the plant is the first class.

Evaluating The Classifier

Now that we have predicted the species of all plants in the test data, we can compare our predicted species with the that plant's actual species.

```
[12] # Create actual english names for the plants for each predicted plant class
     preds = iris.target_names[clf.predict(test[features])]
```

```
[13] # View the PREDICTED species for the first five observations
     preds[0:5]
```

```
array(['setosa', 'setosa', 'setosa', 'setosa', 'setosa'], dtype='<U10')
```

```
[14] # View the ACTUAL species for the first five observations
     test['species'].head()
```

	species
7	setosa
8	setosa
10	setosa
13	setosa
17	setosa

dtype: category

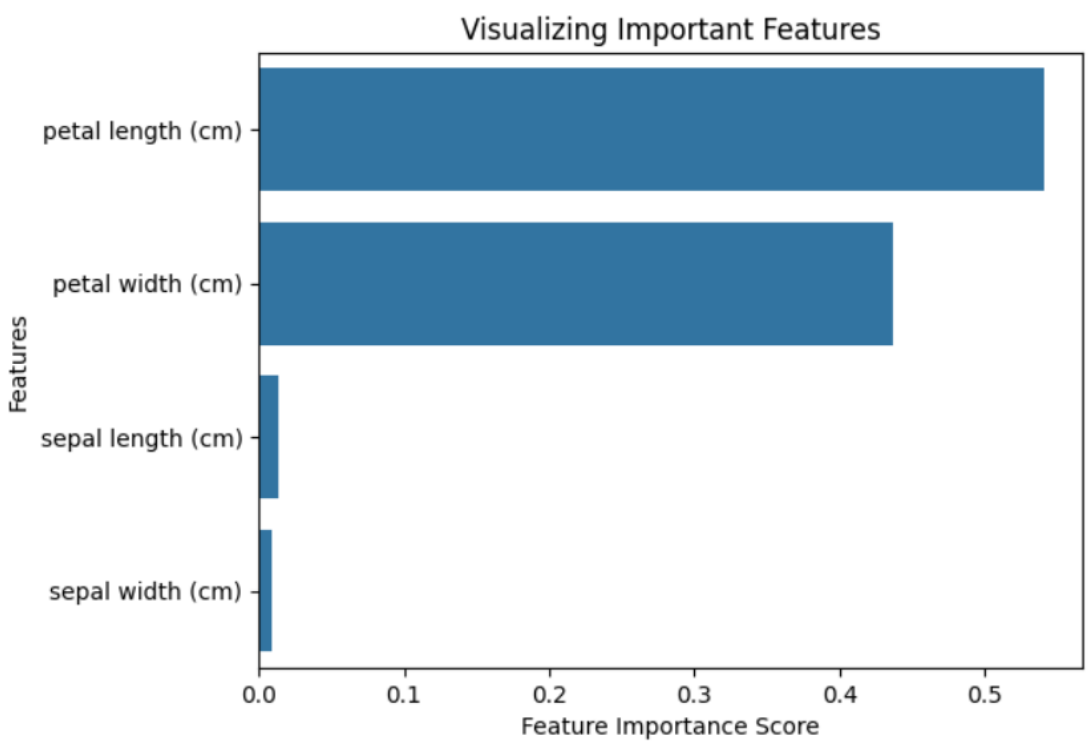
```
[16] # View a list of the features and their importance scores
     #list(zip(train[features], clf.feature_importances_))
     feature_imp = pd.Series(clf.feature_importances_,index=iris.feature_names).sort_values(ascending=False)
     feature_imp
```

	0
petal length (cm)	0.540827
petal width (cm)	0.436527
sepal length (cm)	0.013891
sepal width (cm)	0.008755

dtype: float64

Visualisation

```
[17] import matplotlib.pyplot as plt
     import seaborn as sns

     # Creating a bar plot
     sns.barplot(x=feature_imp, y=feature_imp.index)

     # Add labels to your graph
     plt.xlabel('Feature Importance Score')
     plt.ylabel('Features')
     plt.title("Visualizing Important Features")
     plt.show()
```



```
[18] y_pred = clf.predict(test[features])
     y_pred

     y_test = pd.factorize(test['species'])[0]
     y_test

     #Import scikit-learn metrics module for accuracy calculation
     from sklearn import metrics

     # Model Accuracy, how often is the classifier correct?
     print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.96875
```

C) Subset of 4 Features

Training The Random Forest Classifier

```
[27] # Load scikit's random forest classifier library
     from sklearn.ensemble import RandomForestClassifier

     # Create a random forest Classifier. By convention, clf means 'Classifier'
     # We can specify the number of features using max_features
     clf = RandomForestClassifier(max_features=4, random_state=0)

     # Train the Classifier to take the training features and learn how they relate
     # to the training y (the species)
     clf.fit(train[features], y)
```

```
RandomForestClassifier
RandomForestClassifier(max_features=4, random_state=0)
```

Applying Classifier To Test Data

We leave out a portion of the data so that we have a set of data to test the accuracy of the model

```
[28] # Apply the Classifier we trained to the test data (which, remember, it has never seen before)
     y_pred = clf.predict(test[features])
     y_pred
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 2, 2, 1, 1, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

```
[29] # View the predicted probabilities of the first 10 observations
     clf.predict_proba(test[features])[0:25]
```

```
array([[1.  , 0.  , 0.  ],
       [1.  , 0.  , 0.  ],
       [1.  , 0.  , 0.  ],
       [1.  , 0.  , 0.  ],
       [1.  , 0.  , 0.  ],
       [1.  , 0.  , 0.  ],
       [1.  , 0.  , 0.  ],
       [1.  , 0.  , 0.  ],
       [1.  , 0.  , 0.  ],
       [1.  , 0.  , 0.  ],
       [1.  , 0.  , 0.  ],
       [1.  , 0.  , 0.  ],
       [1.  , 0.  , 0.  ],
       [0.  , 0.62, 0.38],
       [0.  , 1.  , 0.  ],
       [0.  , 0.91, 0.09],
       [0.  , 0.  , 1.  ],
       [0.  , 0.49, 0.51],
       [0.  , 1.  , 0.  ],
       [0.  , 1.  , 0.  ],
       [0.  , 0.  , 1.  ],
       [0.  , 0.  , 1.  ],
       [0.  , 0.  , 1.  ],
       [0.  , 0.  , 1.  ],
       [0.  , 0.  , 1.  ]])
```

The classifier gives a 90% probability the plant belongs to the first class and a 10% the plant belongs to the second class

Because 90 is greater than 10, the classifier predicts the plant is the first class.

Evaluating The Classifier

Now that we have predicted the species of all plants in the test data, we can compare our predicted species with the that plant's actual species

```
[30] # Create actual english names for the plants for each predicted plant class
     preds = iris.target_names[clf.predict(test[features])]
```

```
[31] # View the PREDICTED species for the first five observations
     preds[0:5]
```

```
array(['setosa', 'setosa', 'setosa', 'setosa', 'setosa'], dtype='<U10')
```

```
[32] # View the ACTUAL species for the first five observations
     test['species'].head()
```

	species
7	setosa
8	setosa
10	setosa
13	setosa
17	setosa

dtype: category

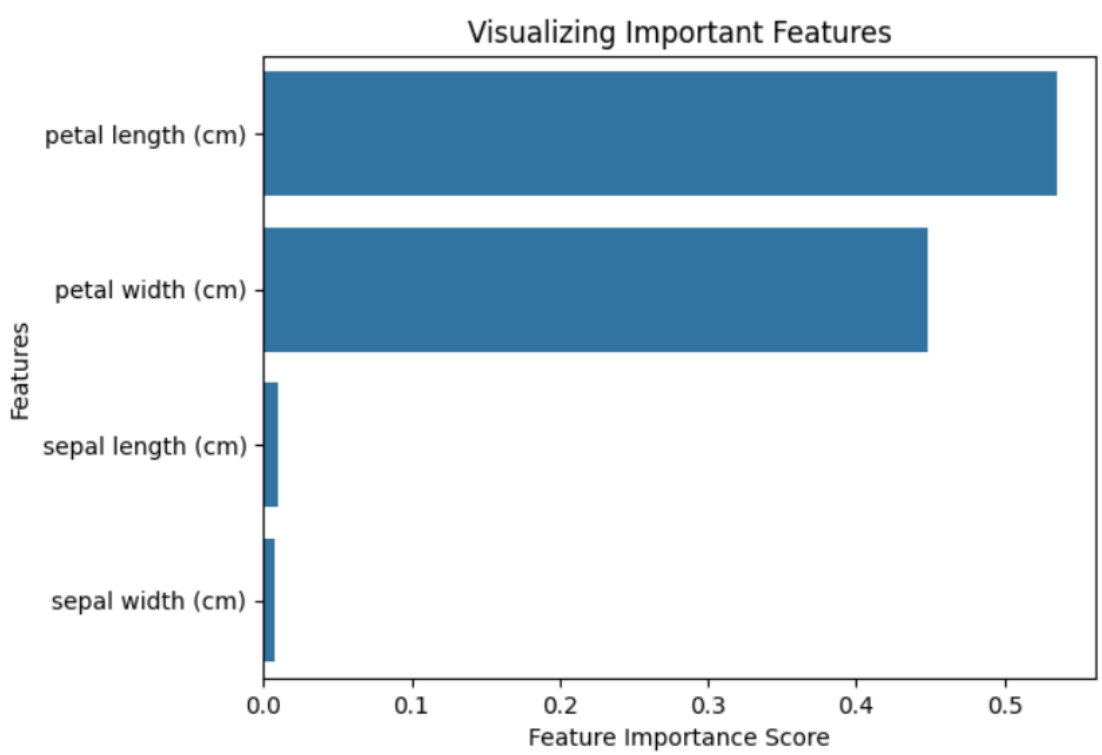
```
[34] # View a list of the features and their importance scores
     #list(zip(train[features], clf.feature_importances_))
     feature_imp = pd.Series(clf.feature_importances_,index=iris.feature_names).sort_values(ascending=False)
     feature_imp
```

	0
petal length (cm)	0.535099
petal width (cm)	0.448016
sepal length (cm)	0.009444
sepal width (cm)	0.007442

dtype: float64

Visualisation

```
[35] import matplotlib.pyplot as plt
     import seaborn as sns

     # Creating a bar plot
     sns.barplot(x=feature_imp, y=feature_imp.index)

     # Add labels to your graph
     plt.xlabel('Feature Importance Score')
     plt.ylabel('Features')
     plt.title("Visualizing Important Features")
     plt.show()
```



```
[36] y_pred = clf.predict(test[features])
     y_pred

     y_test = pd.factorize(test['species'])[0]
     y_test

     #Import scikit-learn metrics module for accuracy calculation
     from sklearn import metrics

     # Model Accuracy, how often is the classifier correct?
     print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.9375
```

Accuracies

Set of 2 features	0.9375
Set of 3 features	0.96875
Set of 4 features	0.9375

- *Which set of features is better? Discuss your observations and justify your answer.*

Subset of 3 Features is found to Perform the Best as the accuracy for 3 features is higher than for 2 or 4 features, this is due to **the optimal balance of features** allowing the model to generalize better without overfitting.

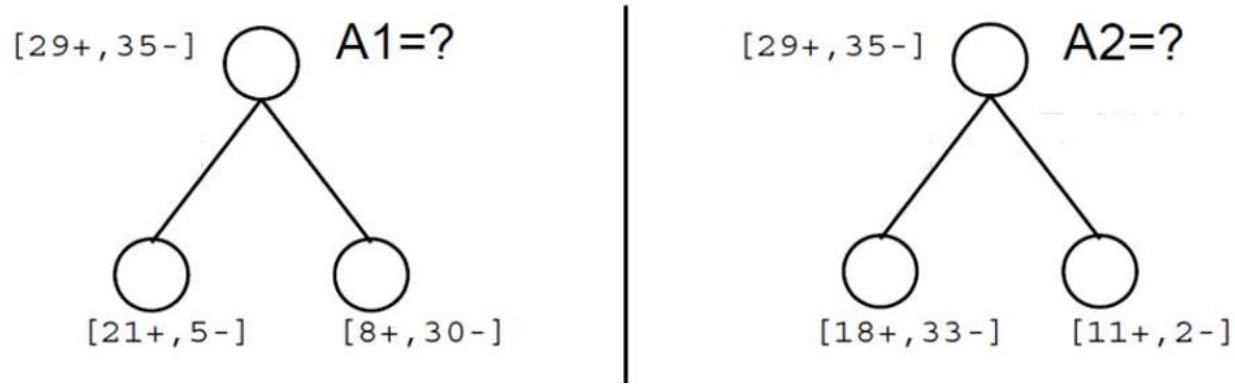
Including all 4 features slightly reduces accuracy, possibly due to overfitting or irrelevant features **diluting the predictive power**. With only 2 features, **the model may lack sufficient information** to fully distinguish between all classes, leading to slightly lower accuracy.

- *What is the largest number of features you may have?*

The largest number of features for the Iris dataset, this is 4 features, as it contains: Sepal length, Sepal width, Petal length and Petal widt

Task 2

You have been given a binary classification problem (positive/negative) where the original dataset contains 29 positive and 35 negative samples. We have 2 features of A1 and A2 which can be used for splitting the data. We would like to build a decision tree and figure below provides the resulting splits for each feature. Which feature would you use to split the data? A1 or A2? Justify your answer.



Ans:

To analyze which feature is best for splitting the data, we first need to calculate the entropy of the Parent Tree

$$Entropy(S) = -p_{\oplus} \log_2 p_{\oplus} - p_{\ominus} \log_2 p_{\ominus}$$

p + = $\frac{positive\ samples}{total\ samples} = \frac{29}{64}$ p - = $\frac{negative\ samples}{total\ samples} = \frac{35}{64}$

H parent = $-\left[\frac{29}{64}\right] \log 2 \left(\frac{29}{64}\right) - \left[\frac{35}{64}\right] \log 2 \left(\frac{35}{64}\right)$

H parent = 0.994

<u>Calculating Entropy of Each Child Node</u>

A1 : Left Child

Entropy ([21+, 5-]) = $-\left[\frac{21}{26}\right] \log 2 \left(\frac{21}{26}\right) - \left[\frac{5}{26}\right] \log 2 \left(\frac{5}{26}\right)$

= -0.807× (-0.308) – 0.192 × (-2.378)

= <u>0.705</u>

A1 : Right Child

Entropy ([8+, 30-]) = $-\left[\frac{8}{38}\right]\log 2\left(\frac{8}{38}\right) - \left[\frac{30}{38}\right]\log 2\left(\frac{30}{38}\right)$

= -0.2105× (-2.247) – 0.789 × (-0.3410)

= <u>0.742</u>

A2 : Left Child

Entropy ([18+, 33-]) = $-\left[\frac{18}{51}\right]\log 2\left(\frac{18}{51}\right) - \left[\frac{33}{51}\right]\log 2\left(\frac{33}{51}\right)$

= -0.352× (-1.504) – 0.647 × (-0.628)

= <u>0.935</u>

A2 : Right Child

Entropy ([11+, 2-]) = $-\left[\frac{11}{13}\right]\log 2\left(\frac{11}{13}\right) - \left[\frac{2}{13}\right]\log 2\left(\frac{2}{13}\right)$

= -0.846× (-0.241) – 0.153 × (-2.700)

= <u>0.616</u>

<u>Avg Entropy of A1</u> = $\left[\frac{26}{64}\right]$ x 0.705 + $\left[\frac{38}{64}\right]$ x 0.742

= <u>0.726</u>

<u>Avg Entropy of A2</u> = $\left[\frac{51}{64}\right]$ x 0.935 + $\left[\frac{13}{64}\right]$ x 0.616

= <u>0.870</u>

Information Gain (IG) = H parent – Avg. Entropy

Gain (S, A1) = <u>0.267</u>

Gain (S, A2) = <u>0.124</u>

The **Information Gain** for **A1** (IG=0.267) is greater than that for **A2** (IG=0.124). Therefore, <u>**A1** is the better feature to split the data.</u>

This split reduces the impurity (entropy) more effectively compared to A2.

Week – 6 – Classification (Linear Classifiers)

Perceptron algorithm is a simple way of implementing a **supervised linear classifier**. Every input of the Perceptron gets multiplied by a weight and then the results get summed up and passed through **an activation function**. In the case of the Perceptron the activation function resembles the **"all-or-none" characteristic of a biological neuron.**

Weights - Each input neuron is associated with a weight, which represents the strength of the connection between the input neuron and the output neuron

Collab Links

https://colab.research.google.com/drive/1T3Yv0pdk9UFPYgeAr5Gcrwp5LDSBYTCS?usp=sharing

Task 1

For this task, download data_Perceptron.txt from Blackboard. This is a simple data set that contains 1000 entries with 2 features and corresponding class labels (0 or 1) for each entry. Snapshot of data (first two columns are features, and last column is the class label). Apply the Perceptron algorithm to obtain a linear classifier and plot the results.

- Is the data linearly separable? Justify your answer.
- If linearly inseparable, what is the percentage of misclassified points using your linear classifier??

Tip: compute the total number of misclassified points (classified as 0 where it should be 1, or classified as 1 where it should be zero), and then divide by total number of points (i.e. 1000).

Use the Python script for this task, but it needs to be modified

```
X = data[:, 0:2]  # Features (all columns except last)
Y = data[:, 2]   # Labels (last column)
```

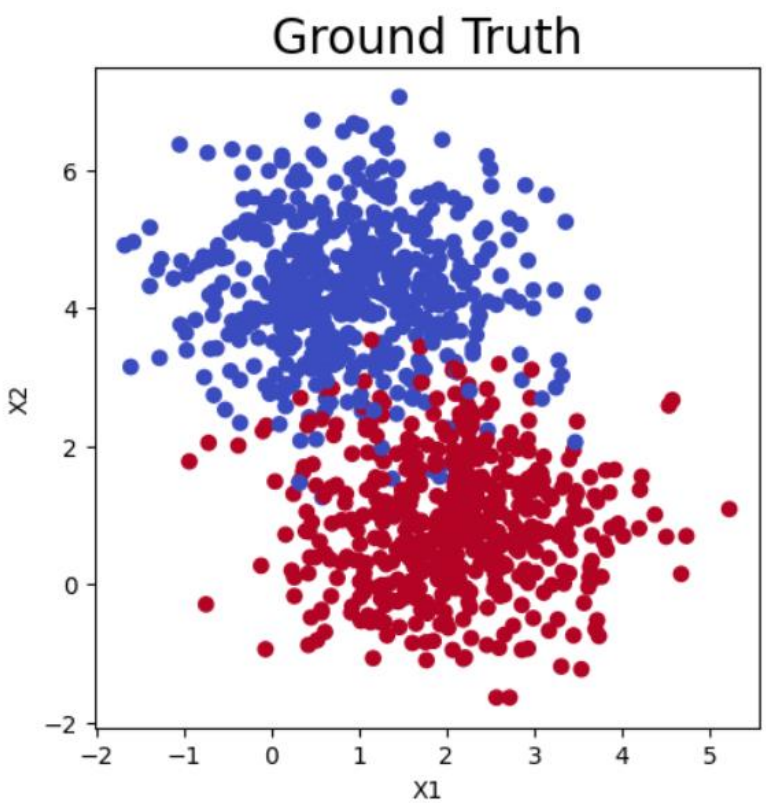
Ans:

```
[2] import numpy as np
    import matplotlib.pyplot as plt
    from google.colab import drive
    drive.mount('/content/gdrive')

    # Load data from data_Perceptron.txt
    data = np.loadtxt('/content/gdrive/My Drive/Colab Notebooks/UWL/ML_L6/data_perceptron.txt')

    X = data[:, 0:2]  # Features (all columns except last)
    Y = data[:, 2]   # Labels (last column)

    # Plot the Ground Truth
    fig, ax = plt.subplots(1, 1, figsize=(5, 5))
    ax.scatter(X[:, 0], X[:, 1], c=Y, cmap='coolwarm')  # Use coolwarm colormap for better visual
    ax.set_title('Ground Truth', fontsize=20)
    plt.xlabel('X1')
    plt.ylabel('X2')
    plt.show()

    # Add a bias to the X vector
    X_bias = np.ones([X.shape[0], 3])
    X_bias[:, 1:3] = X

    # Initialize weight vector with zeros
    w = np.zeros([3, 1])

    # Define the activation function that returns either 1 or 0
    def activation(x):
        return 1 if x >= 0 else 0
```

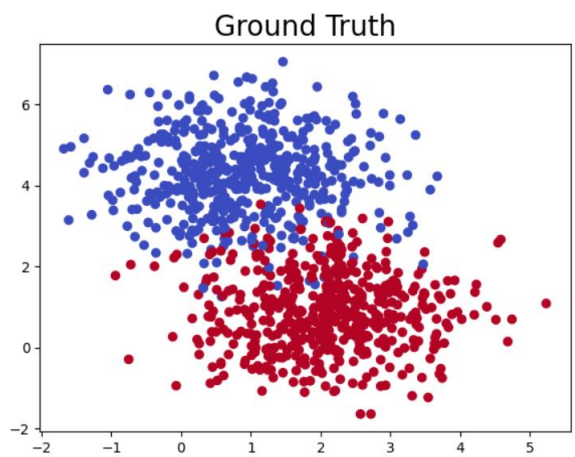
```python
# Perceptron Algorithm
for _ in range(10):  # Training for 10 iterations
    for i in range(X_bias.shape[0]):
        y = activation(w.transpose().dot(X_bias[i, :]))  # Predicted class
        # Update weights if misclassified
        w = w + ((Y[i] - y) * X_bias[i, :]).reshape(w.shape[0], 1)

# Print learned weights
print('w0 = ', w[0])
print('w1 = ', w[1])
print('w2 = ', w[2])

# Calculate the class of the data points with the weight vector
result = [w.transpose().dot(x) for x in X_bias]
result_class = [activation(w.transpose().dot(x)) for x in X_bias]

# Misclassified points calculation
misclassified_points = sum(1 for r, y in zip(result_class, Y) if r != y)
misclassification_rate = misclassified_points / len(Y)
print(f'Misclassification rate: {misclassification_rate * 100:.2f}%')

# Calculate unit vector
def calc_unit_vector(x):
    return x.transpose() / np.sqrt(x.transpose().dot(x))

# Calculate hyperplane
def calc_hyperplane(X, w):
    return np.ravel([-(w[0] + x * w[1]) / w[2] for x in X])
```

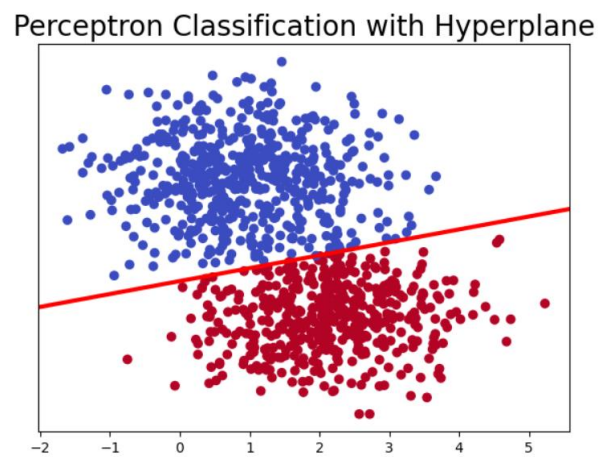
```python
# Normalize weight vector
w = calc_unit_vector(w).transpose()

# Plot the Ground Truth and Perceptron Classification with Hyperplane
fig, ax = plt.subplots(1, 2, figsize=(15, 5))
ax[0].scatter(X[:, 0], X[:, 1], c=Y, cmap='coolwarm')  # Use coolwarm colormap for better visual
ax[0].set_title('Ground Truth', fontsize=20)

ax[1].scatter(X[:, 0], X[:, 1], c=result_class, cmap='coolwarm')  # Use coolwarm colormap for better visual
ax[1].plot([-20, 20], calc_hyperplane([-20, 20], w), lw=3, c='red')  # Red line for hyperplane
ax[1].set_xlim(ax[0].get_xlim())
ax[1].set_ylim(ax[0].get_ylim())
ax[1].set_yticks([])
ax[1].set_title('Perceptron Classification with Hyperplane', fontsize=20)
plt.show()
```

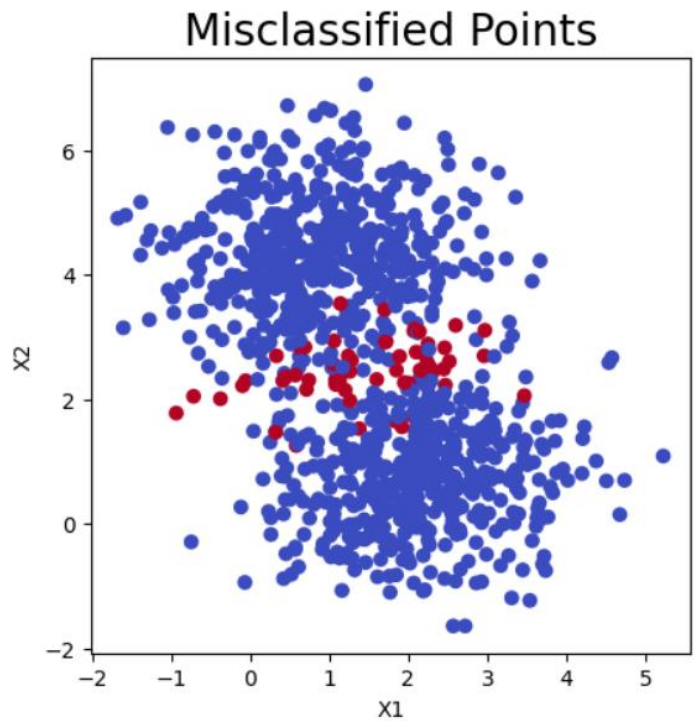
```
# Calculate the misclassified points
misclassified = [1 if predicted != actual else 0 for predicted, actual in zip(result_class, Y)]

# Plot the misclassified points
fig, ax = plt.subplots(1, 1, figsize=(5, 5))
ax.scatter(X[:, 0], X[:, 1], c=misclassified, cmap='coolwarm')  # Use coolwarm colormap for better visual
ax.set_title('Misclassified Points', fontsize=20)
plt.xlabel('X1')
plt.ylabel('X2')
plt.show()

# Calculate the percentage of misclassified points
misclassification_rate = sum(misclassified) / len(misclassified)
print(f'Misclassification Rate: {misclassification_rate * 100:.2f}%')
```




```
w0 =  [25.]
w1 =  [4.85]
w2 =  [-15.21]
Misclassification rate: 5.90%
```







Based on our output we can hypothesize that, No, the data **is NOT perfectly linearly separable**. This is evident from:

- The overlapping region between the two classes in the middle.
- The plot of misclassified points, which shows several data points that cannot be correctly classified by a linear decision boundary.

The percentage of misclassified points is 5.90%

Week – 7 – Classification (Support Vector Machines)

Support vector machines (SVMs) are a particularly powerful and flexible class of supervised algorithms for both classification and regression.

A **SVM** is widely used for both linear and nonlinear classification, as well as regression and outlier detection tasks. SVMs are **highly adaptable**, making them suitable for various applications such as text classification, image classification, spam detection, handwriting identification, gene expression analysis, face detection, and anomaly detection.

SVMs are particularly effective because they **focus on finding the maximum separating hyperplane between the different classes** in the target feature.

Discriminative classification where simply find a line or curve or manifold that divides the classes from each other.

- Line / Curve - 2D
- Manifold - 2+ D

In SVMs, **the line that maximizes this margin is the one we will choose as the optimal model**. Support vector machines are an example of such a **maximum margin estimator**.

Collab Links

https://colab.research.google.com/drive/16oMxUaedrTE-voGHB_TV5uGAKwx0PIWj?usp=sharing

Task-1

For this task, download data_SVM.txt from Blackboard. This is a simple data set that contains 1000 entries with 2 features and corresponding class labels (0 or 1) for each entry.

Snapshot of data (first two columns are features, and last column is the class label). Use the Python script for this task, but it needs to be modified to load the data. You can load the data as follows: Apply the SVM algorithm to obtain a classifier and plot the results.

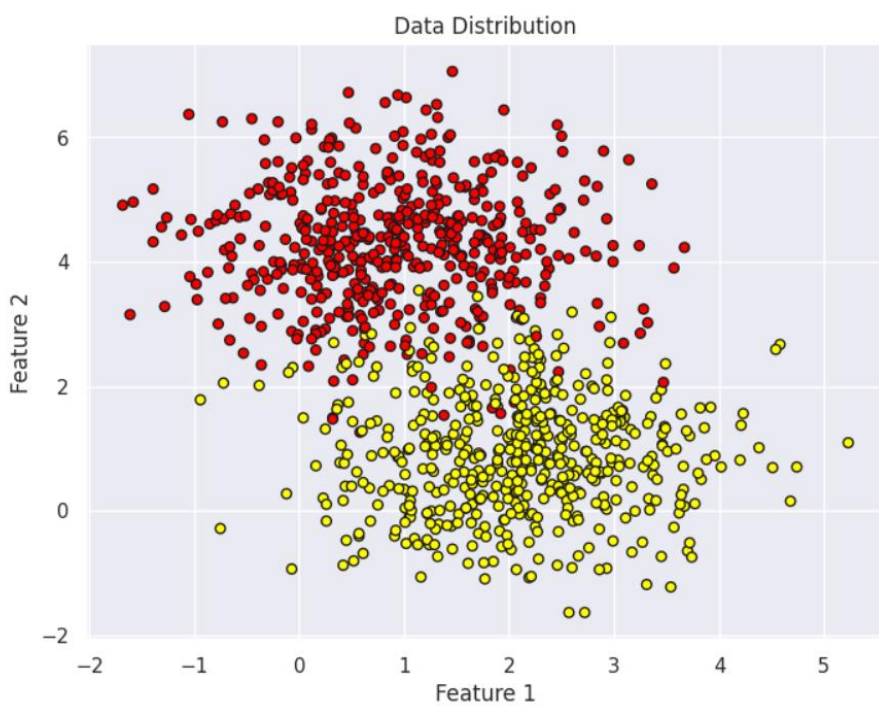
- What is the percentage of misclassified points using your SVM classifier?
- Try the algorithm for a range of different values of C parameter [0.01, 0.1, 1.0], and provide the results together with your understanding of the influence of the C parameter.
- Compare your SVM results with those from the Perceptron in week-6.

Ans:

```
[2] import numpy as np
    import matplotlib.pyplot as plt
    from sklearn.svm import SVC
    from sklearn.metrics import accuracy_score
    import seaborn as sns; sns.set()
    from google.colab import drive
    drive.mount('/content/gdrive')

    # Load data from data_SVM.txt
    data = np.loadtxt('/content/gdrive/My Drive/Colab Notebooks/UWL/ML_L6/data_SVM.txt')
    # Load the dataset
    X = data[:, :2]  # First two columns are features
    y = data[:, 2]   # Last column is the class label

    # Plot the raw dataset
    plt.figure(figsize=(8, 6))
    plt.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap='autumn', edgecolors='k')
    plt.title("Data Distribution")
    plt.xlabel("Feature 1")
    plt.ylabel("Feature 2")
    plt.show()

    # Define a function to plot SVM results
    def plot_svm_decision_boundary(X, y, model, ax=None, plot_support=True):
        if ax is None:
            ax = plt.gca()
        xlim = ax.get_xlim()
        ylim = ax.get_ylim()
```

```
# Create grid for plotting decision function
x = np.linspace(xlim[0], xlim[1], 200)
y_grid = np.linspace(ylim[0], ylim[1], 200)
Y, X_grid = np.meshgrid(y_grid, x)
xy = np.vstack([X_grid.ravel(), Y.ravel()]).T
P = model.decision_function(xy).reshape(X_grid.shape)

# Plot decision boundary and margins
ax.contour(X_grid, Y, P, colors='k', levels=[-1, 0, 1], alpha=0.5, linestyles=['--', '-', '--'])

# Plot support vectors
if plot_support:
    ax.scatter(model.support_vectors_[:, 0], model.support_vectors_[:, 1],
               s=300, linewidth=1, facecolors='none', edgecolors='k')
ax.set_xlim(xlim)
ax.set_ylim(ylim)
```

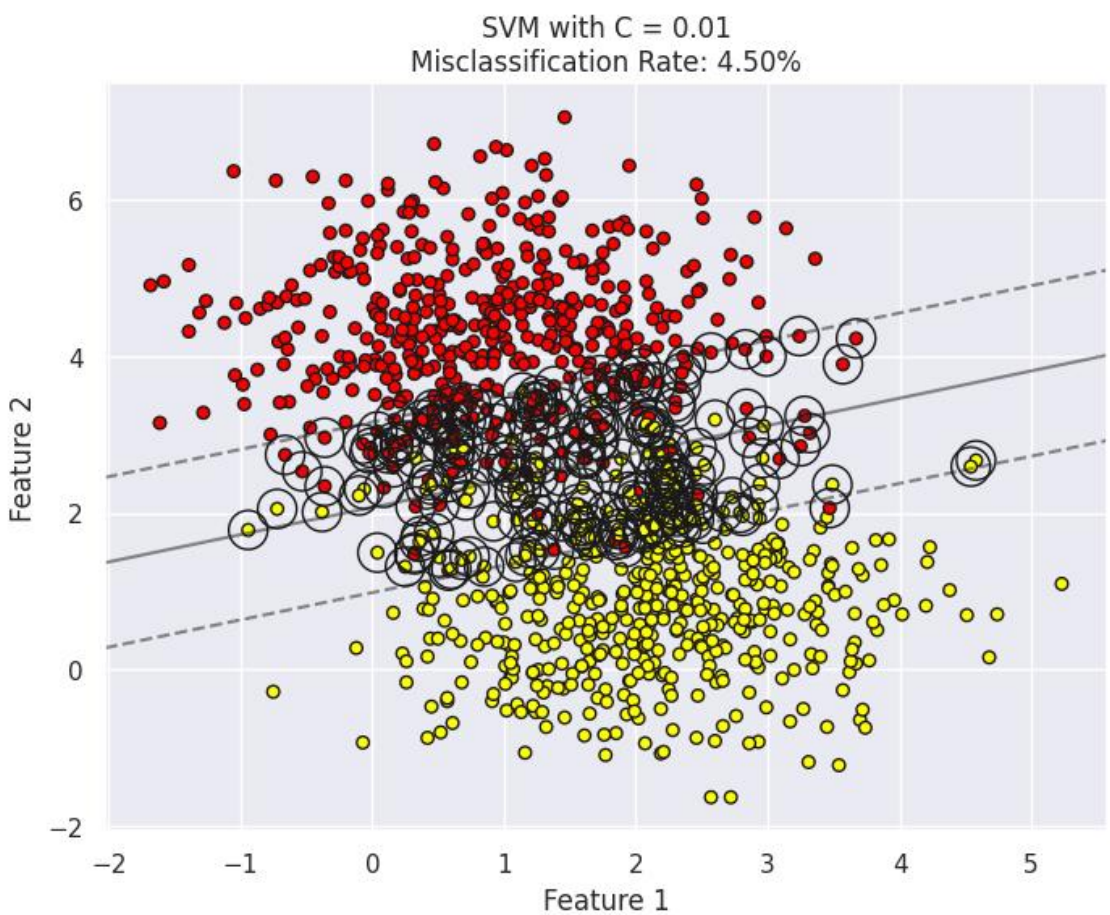
```python
# Train and evaluate SVM for different values of C
C_values = [0.01, 0.1, 1.0]
results = []

plt.figure(figsize=(18, 6))
for i, C in enumerate(C_values, 1):
    # Train the SVM
    model = SVC(kernel='linear', C=C)
    model.fit(X, y)

    # Predict and calculate misclassification rate
    y_pred = model.predict(X)
    misclassification_rate = 1 - accuracy_score(y, y_pred)
    results.append((C, misclassification_rate))

    # Plot the decision boundary
    ax = plt.subplot(1, 3, i)
    ax.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap='autumn', edgecolors='k')
    plot_svm_decision_boundary(X, y, model, ax)
    ax.set_title(f"SVM with C = {C}\nMisclassification Rate: {misclassification_rate*100:.2f}%")
    ax.set_xlabel("Feature 1")
    ax.set_ylabel("Feature 2")

plt.tight_layout()
plt.show()
```



Tuning the SVM when c value is:

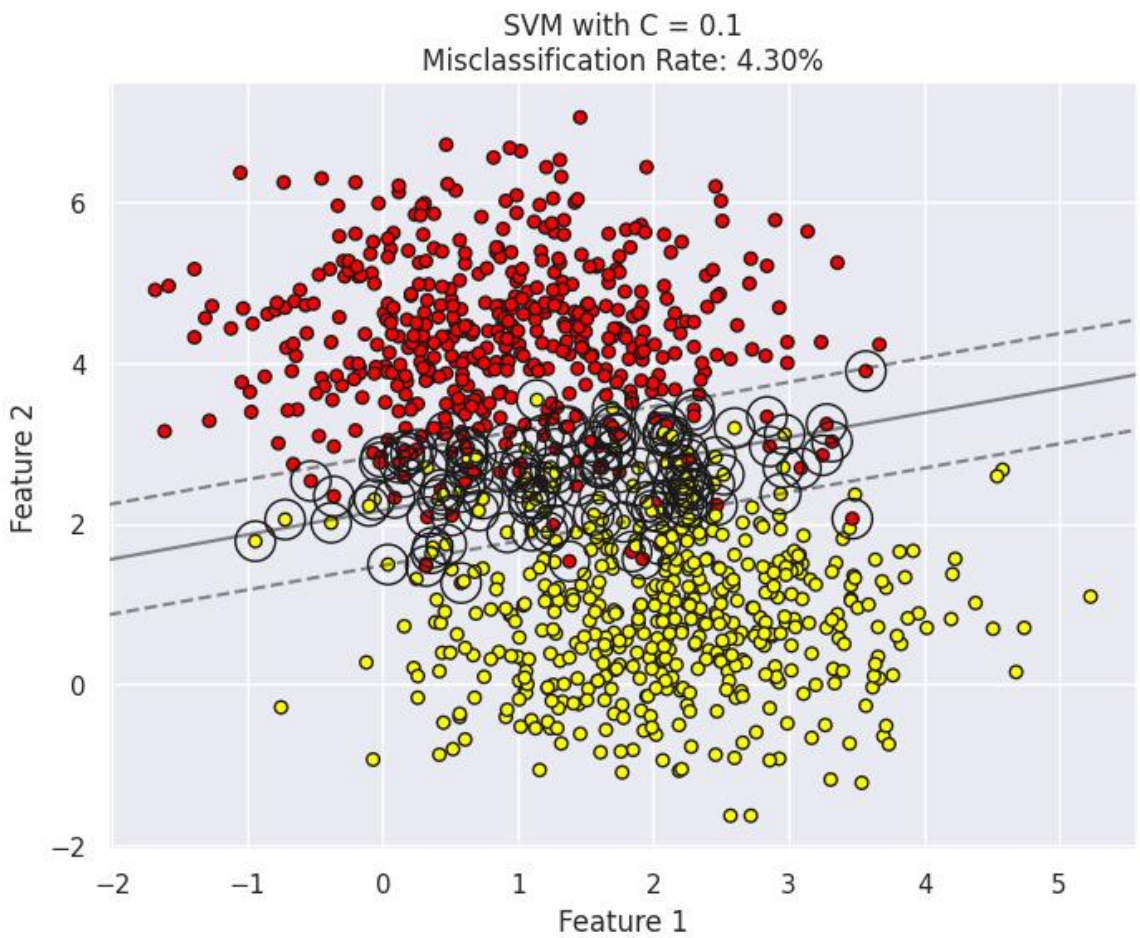
<u>c = 0.01</u>

```
# SVM with C = 0.01
model = SVC(kernel='linear', C=0.01)
model.fit(X, y)
y_pred = model.predict(X)
misclassification_rate = 1 - accuracy_score(y, y_pred)
plt.figure(figsize=(8, 6))
plt.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap='autumn', edgecolors='k')
plot_svm_decision_boundary(X, y, model)
plt.title(f"SVM with C = 0.01\nMisclassification Rate: {misclassification_rate*100:.2f}%")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.show()
```



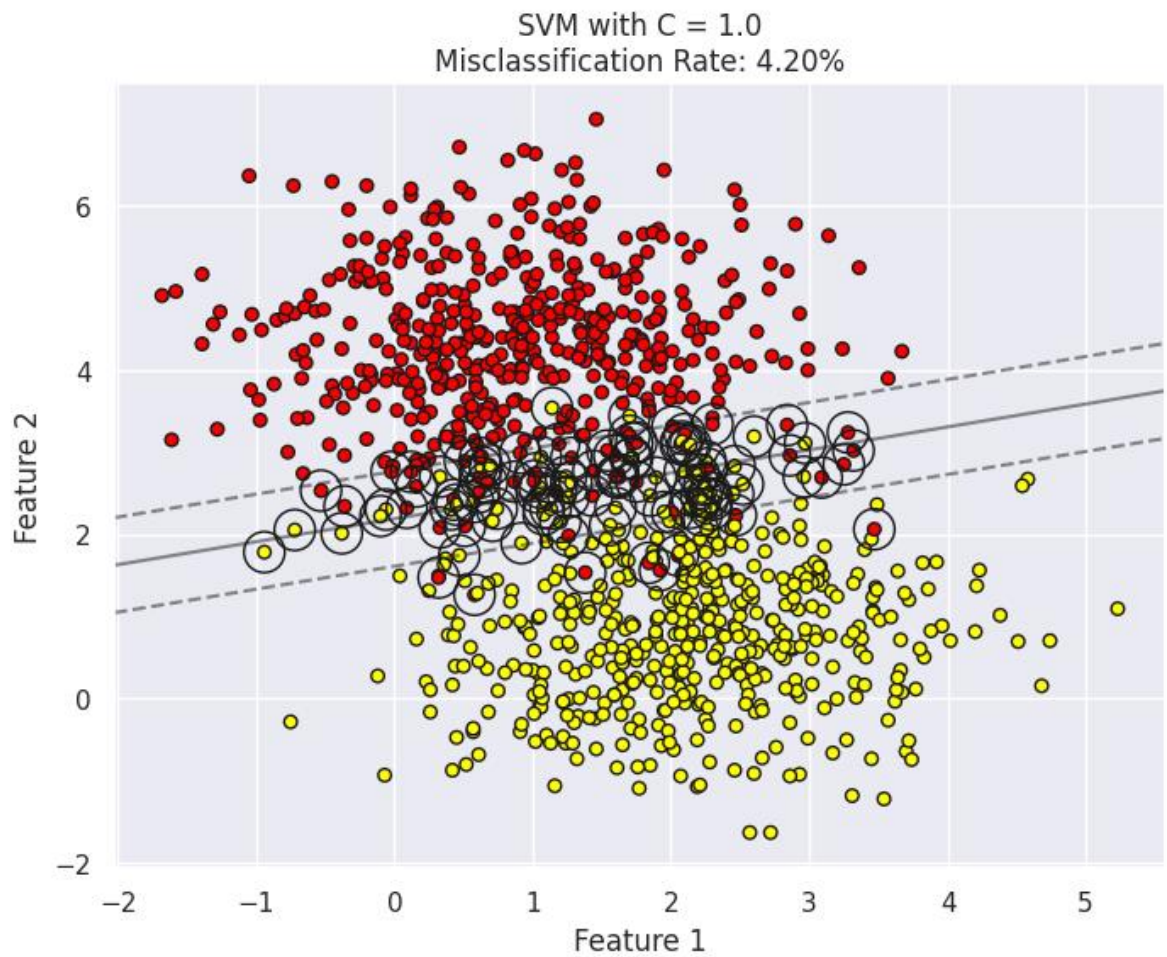
c = 0.1

```
# SVM with C = 0.1
model = SVC(kernel='linear', C=0.1)
model.fit(X, y)
y_pred = model.predict(X)
misclassification_rate = 1 - accuracy_score(y, y_pred)
plt.figure(figsize=(8, 6))
plt.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap='autumn', edgecolors='k')
plot_svm_decision_boundary(X, y, model)
plt.title(f"SVM with C = 0.1\nMisclassification Rate: {misclassification_rate*100:.2f}%")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.show()
```



c = 1.0

```
# SVM with C = 1.0
model = SVC(kernel='linear', C=1.0)
model.fit(X, y)
y_pred = model.predict(X)
misclassification_rate = 1 - accuracy_score(y, y_pred)
plt.figure(figsize=(8, 6))
plt.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap='autumn', edgecolors='k')
plot_svm_decision_boundary(X, y, model)
plt.title(f"SVM with C = 1.0\nMisclassification Rate: {misclassification_rate*100:.2f}%")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.show()
```



Result Summary

C value	Misclassification Rate
0.01	4.50 %
0.1	4.30 %
1.0	4.20 %

Analysis

- Smaller C values (e.g., C = 0.01) allow for larger margins, resulting in higher flexibility but potentially more misclassification.
- Larger C values (e.g., C = 1.0) enforce stricter margins and aim to minimize classification error on the training set, but may lead to overfitting.

Comparison with Perceptron

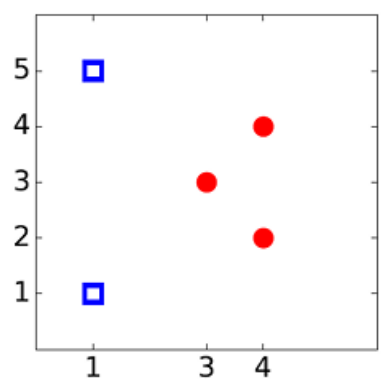
SVM outperforms the Perceptron with lower misclassification rates across all C values.

Perceptron Misclassification Rate: 5.90%

Task-2

Consider we are learning a linear SVM for a training dataset shown in a table below, described with two features x1 and x2, and binary target y. The classifier takes the form f(x; w1; w2; b) = sign(w1x1 + w2x2 + b):

x_1	x_2	y
1	1	-1
1	5	-1
3	3	+1
4	2	+1
4	4	+1



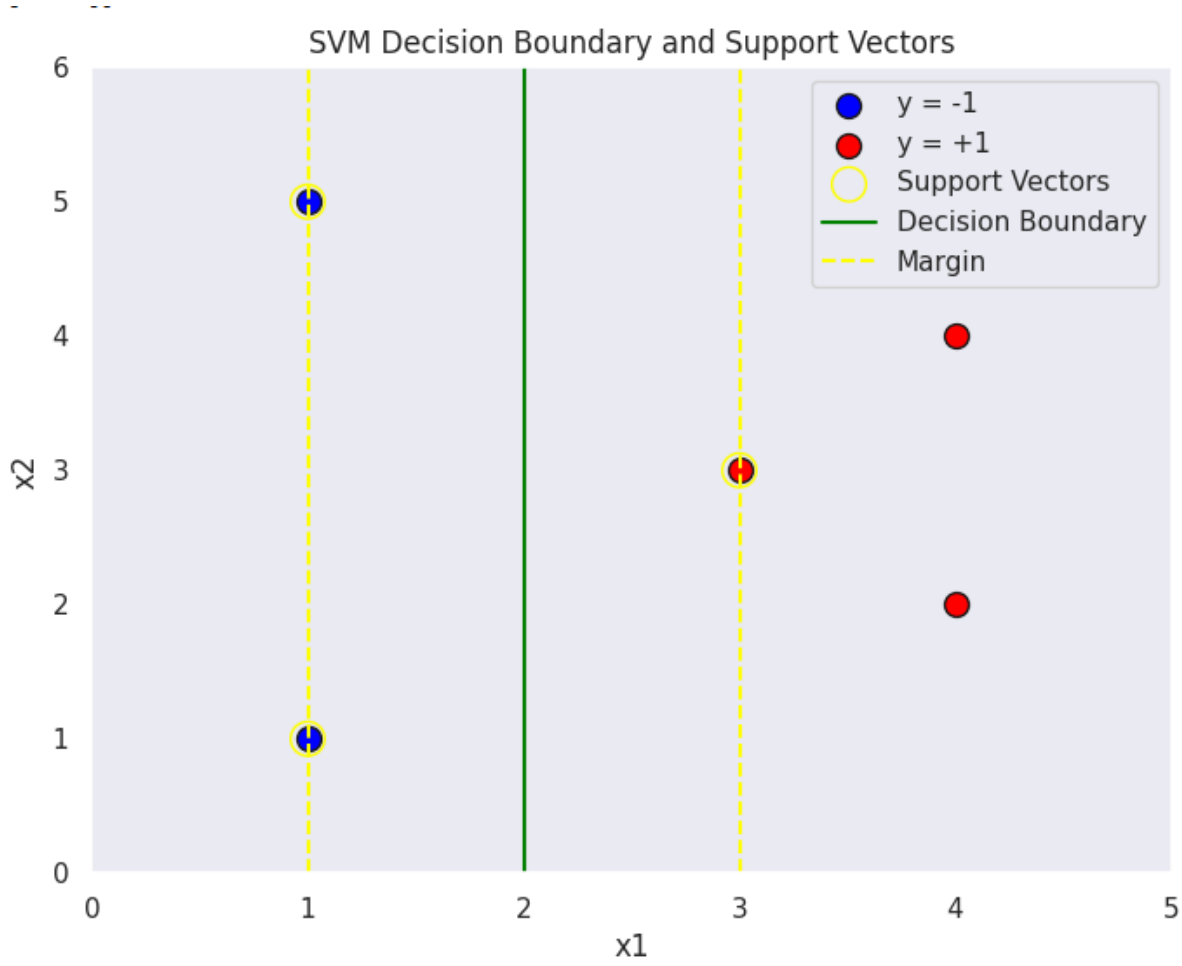
- Draw the decision boundary of the trained SVM, and identify the support vectors.
- Calculate the parameters of the trained SVM (i.e., w1, w2, b).

Ans:

```
[9] import numpy as np
    import matplotlib.pyplot as plt
    from sklearn.svm import SVC

    # Training dataset
    X = np.array([[1, 1], [1, 5], [3, 3], [4, 2], [4, 4]])
    y = np.array([-1, -1, 1, 1, 1])

    # Train a linear SVM
    model = SVC(kernel='linear', C=1.0)
    model.fit(X, y)

    # Get model parameters
    w = model.coef_[0]  # Coefficients (w1, w2)
    b = model.intercept_[0]  # Bias term
    print(f"Model Parameters: w1 = {w[0]:.2f}, w2 = {w[1]:.2f}, b = {b:.2f}")

    # Find support vectors
    support_vectors = model.support_vectors_
    print(f"Support Vectors:\n{support_vectors}")

    # Plot the data points
    plt.figure(figsize=(8, 6))
    plt.scatter(X[y == -1][:, 0], X[y == -1][:, 1], color='blue', s=100, edgecolors='k', label='y = -1')
    plt.scatter(X[y == 1][:, 0], X[y == 1][:, 1], color='red', s=100, edgecolors='k', label='y = +1')
```

```
# Plot support vectors
plt.scatter(support_vectors[:, 0], support_vectors[:, 1], s=200, facecolors='none', edgecolors='yellow', label='Support Vectors')

# Plot decision boundary
x_boundary = -b / w[0]  # Since decision boundary is w1*x1 + w2*x2 + b = 0
plt.axvline(x=x_boundary, color='green', linestyle='-', label='Decision Boundary')

# Plot margin lines
margin = 1 / np.sqrt(np.sum(model.coef_ ** 2))
plt.axvline(x=x_boundary + margin / abs(w[0]), color='yellow', linestyle='--', label='Margin')
plt.axvline(x=x_boundary - margin / abs(w[0]), color='yellow', linestyle='--')

# Annotate plot
plt.xlim(0, 5)
plt.ylim(0, 6)
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('SVM Decision Boundary and Support Vectors')
plt.legend()
plt.grid()
plt.show()
```

```
Model Parameters: w1 = 1.00, w2 = 0.00, b = -2.00
Support Vectors:
[[1. 1.]
 [1. 5.]
 [3. 3.]]
```


SVM Decision Boundary and Support Vectors
y = -1
y = +1
Support Vectors
Decision Boundary
Margin
x1
x2
0
1
2
3
4
5
6

Week – 8 – Classification (Neural Networks)

To **avoid Overfitting** it is common practice when performing a (supervised) machine learning experiment to hold out part of the available data as a test set, Knowledge about the **test set can "leak" into the model** and evaluation metrics no longer report on generalization performance.

To solve this problem, another part of the dataset can be held out as **validation set**: training proceeds on the training set, after which evaluation is done on the validation set, and when the experiment seems to be successful, final evaluation can be done on the test set.

However, by partitioning the available data into three sets, we drastically reduce the number of samples which can be used for learning the model, and the results can depend on a particular random choice for the pair of sets.

A solution to this problem is a procedure called **cross-validation** (CV). A test set should still be held out for final evaluation, but the validation set is no longer needed when doing CV, this approach can be **computationally expensive**, but does not waste too much data.

Collab Links

https://colab.research.google.com/drive/1PYvrSXa8L6Y8OVSRMmK75qBN4gBgxWHp?usp=sharing

Task 1

Explain how a Multi-layer Perceptron can address the limitation of a single-layer Perceptron.

Ans: A Multi-layer Perceptron (MLP) addresses the limitations of a Single-layer Perceptron (SLP) by **introducing hidden layers and non-linear activation functions**, allowing it to model more complex relationships within the data.

An SLP consists of only an input layer and an output layer, and it can only solve problems that are **linearly separable**. This means that the data must be separable by a straight line (or a hyperplane). This limitation arises because the SLP **applies a linear transformation to the data**, **restricting its ability to capture non-linear patterns**.

An MLP, on the other hand, consists of an input layer, **one or more hidden layers**, and an output layer. The hidden layers use **non-linear activation functions** (such as ReLU

or sigmoid) to transform the input data, enabling the network to learn non-linear decision boundaries, So **SLP is confined to linearly separable problems**, an MLP can solve non-linearly separable problems by using multiple hidden layers and non-linear activation functions

Task 2

The script applies Multi-layer Perceptron Classifier (Neural Network) to the Iris dataset. Search for the parameter hidden_layer_sizes in the code. You can use this parameter to define the number of neurons and layers in the network architecture:

ith element represents the number of neurons in the ith hidden layer.

Try a few different configurations for the network architecture and discuss your observations in the logbook.

- How does the number of neurons and hidden layers affect the performance of the network?
- Can you find any architecture that yields 100% accuracy on test set?

Ans:

```
[1] from sklearn.datasets import load_iris
    from sklearn.model_selection import train_test_split
    from sklearn.neural_network import MLPClassifier
    from sklearn.metrics import accuracy_score, confusion_matrix
    import matplotlib.pyplot as plt
    from sklearn.utils.multiclass import unique_labels
    import numpy as np

    # Load Iris Dataset
    iris = load_iris()
    X = iris.data
    y = iris.target
    class_names = iris.target_names

    # Split dataset into training and testing sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=3)

    # Experiment with different network architectures
    hidden_layer_configurations = [
        (2,),            # Single hidden layer with 2 neurons
        (4, 4),          # Two hidden layers with 4 neurons each
        (10, 5),         # Two hidden layers with 10 and 5 neurons
        (20, 10, 5),     # Three hidden layers
        (50, 50),        # Two hidden layers with 50 neurons each
    ]
```

```python
for config in hidden_layer_configurations:
    print(f"\nTesting architecture: {config}")

    # Initialize the MLP Classifier with the current architecture
    mlp = MLPClassifier(hidden_layer_sizes=config, max_iter=10000, learning_rate='adaptive', random_state=1)

    # Train the model
    mlp.fit(X_train, y_train)

    # Predict on test data
    y_pred = mlp.predict(X_test)

    # Calculate accuracy
    accuracy = accuracy_score(y_test, y_pred)
    print(f"Accuracy on test set: {accuracy * 100:.2f}%")
```

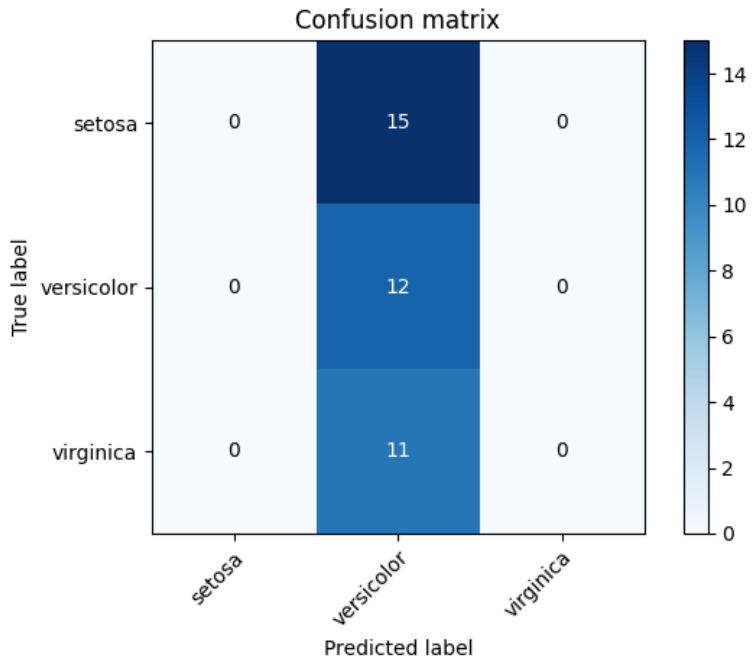
Plotting Confusion Matrix

```python
# Confusion matrix
def plot_confusion_matrix(y_true, y_pred, classes, cmap=plt.cm.Blues):
    cm = confusion_matrix(y_true, y_pred)
    classes = classes[unique_labels(y_true, y_pred)]
    fig, ax = plt.subplots()
    im = ax.imshow(cm, interpolation='nearest', cmap=cmap)
    ax.figure.colorbar(im, ax=ax)
    ax.set(xticks=np.arange(cm.shape[1]),
           yticks=np.arange(cm.shape[0]),
           xticklabels=classes, yticklabels=classes,
           title='Confusion matrix',
           ylabel='True label',
           xlabel='Predicted label')
    plt.setp(ax.get_xticklabels(), rotation=45, ha="right", rotation_mode="anchor")
    fmt = 'd'
    thresh = cm.max() / 2.
    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            ax.text(j, i, format(cm[i, j], fmt),
                    ha="center", va="center",
                    color="white" if cm[i, j] > thresh else "black")
    fig.tight_layout()
    return ax

# Plot confusion matrix
plot_confusion_matrix(y_test, y_pred, class_names)
plt.show()
```

Hidden Layers Used

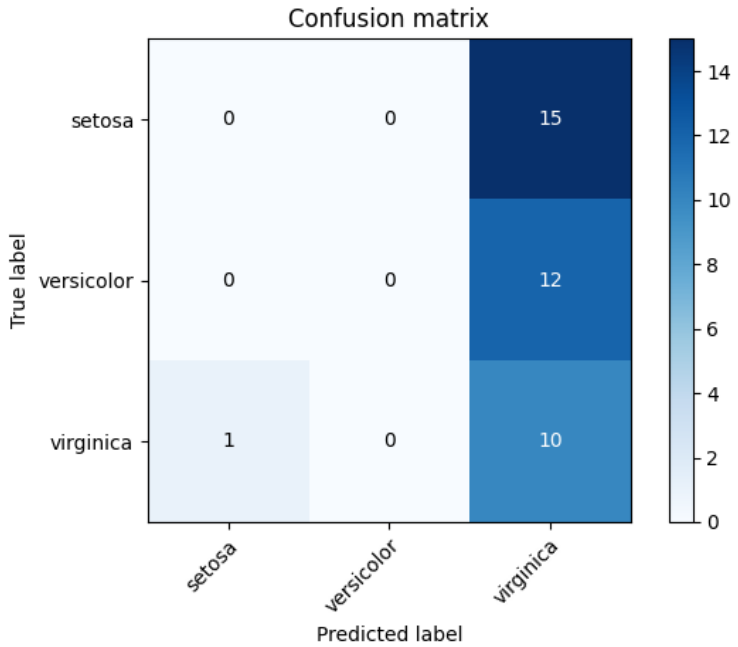
- (2,) – Single Layer with 2 neurons (Underfitting)

```
Testing architecture: (2,)
Accuracy on test set: 31.58%
```



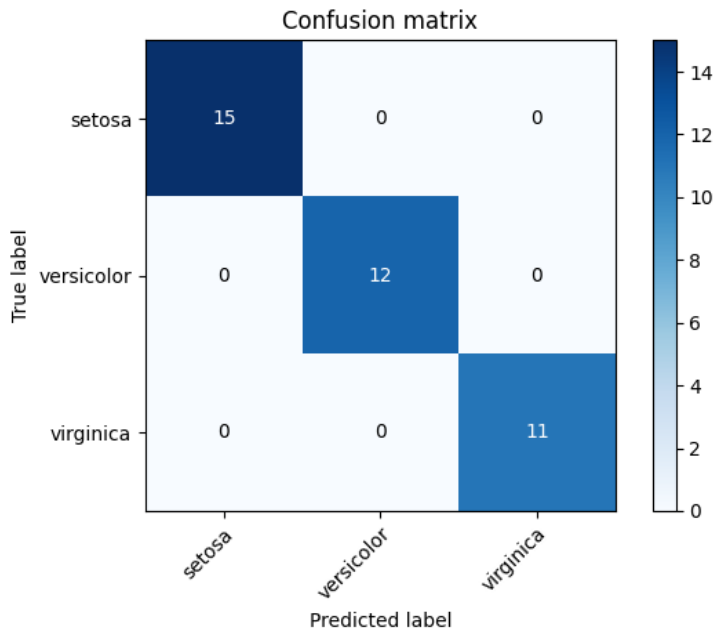
- (4,4) – Two Hidden layers with 4 neurons each (Underfitting)

```
Testing architecture: (4, 4)
Accuracy on test set: 26.32%
```



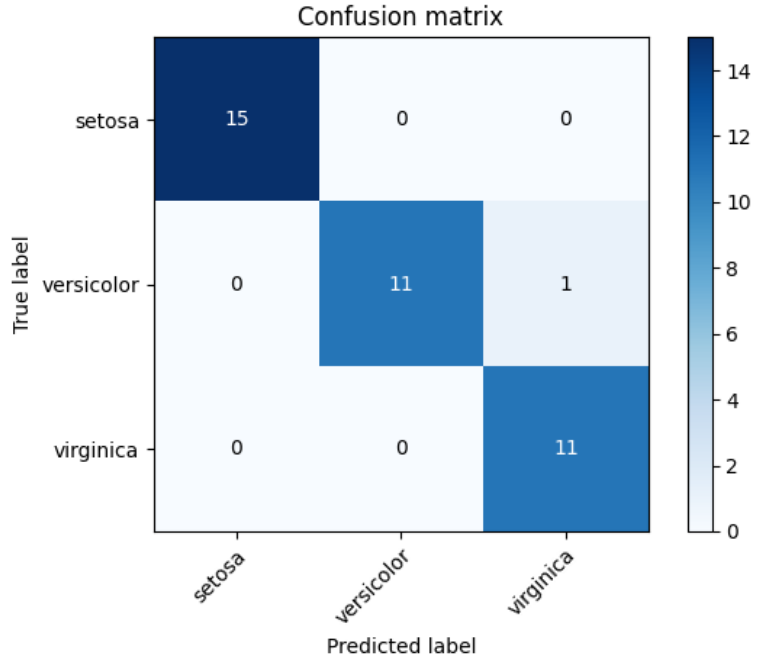
- (10,5) – Two Hidden layers with 10 and 5 neurons **(Perfect performance)**

```
Testing architecture: (10, 5)
Accuracy on test set: 100.00%
```



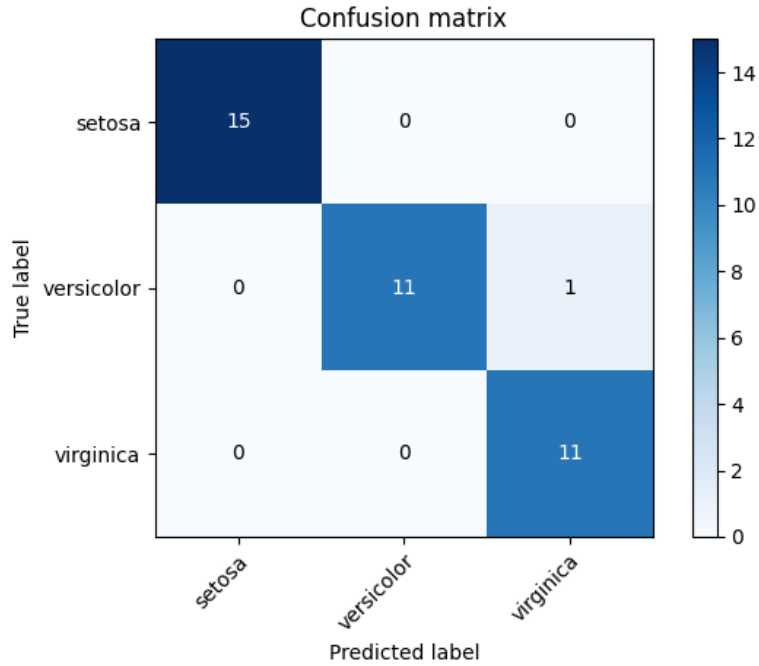
- (20,10,5) – Three Hidden Layers with 20 10 and 5 neurons each (Good Balance)

```
Testing architecture: (20, 10, 5)
Accuracy on test set: 97.37%
```



- (50,50) – Three Hidden Layers with 20 10 and 5 neurons each (Overfitting)

```
Testing architecture: (50, 50)
Accuracy on test set: 97.37%
```



Observations

- Very Small architectures (2,) **may result in underfitting** and low accuracy.
- Medium architectures (10, 5) or (20, 10, 5) are likely to **perform well**
- Very large architectures (50,50) **might overfit**, especially if the dataset is small like Iris.

Increasing the number of layers can improve model capacity but may also increase training time.
More neurons in each layer allow the network to learn complex relationships, but **excessive neurons may lead to overfitting**.

The architecture that yields **100% accuracy on the Iris test set is (10,5).**

Week – 9 – Classification (Image Classification)

The KNN classifier consists of these stages:

- During training, the classifier takes the training data and simply remembers it
- During testing, kNN classifies every test image by comparing to all training images and transfering the labels of the k most similar training examples
- The value of k is cross-validated

Perform k-fold cross validation to find the best value of k.

For each possible value of k, run the k-nearest-neighbor algorithm num_folds times, where in each case you use all but one of the folds as training data and the last fold as a validation set.

Collab Links

https://colab.research.google.com/drive/1riNGGTXW6dC_TBNiC0D-8GseAuhV1wWD?usp=sharing

Task 1

Describe one method for setting Hyperparameters in k-NN classification method.

Ans: One effective method for setting hyperparameters in the k-Nearest Neighbors (k-NN) classification method is **Cross-Validation**. In CV we first,

1. Split the Training Data into Folds we divide the training dataset into k folds (e.g., 5 or 10 folds).
2. Each fold acts as a validation set, while the remaining folds form the training set for that iteration.
3. Choose a range of possible values for the hyperparameter k (e.g., 1, 3, 5, 7, etc.).
4. For each value of k, train the k-NN model using the training folds and calculate its accuracy on the validation fold.
5. Compute the Average Accuracy for each value of k, compute the average accuracy across all folds and select the best k.

Task 2

Based on the cross-validation results in the Introduction exercises above, choose the best value for k in k-NN. Justify your choice.
Then retrain the image classifier using half the CIFAR10 training data, and test it on all the test data.
What is the level accuracy?

Ans:

```
[ ] from google.colab import drive
    drive.mount('/content/gdrive')
    FolderName = '/content/gdrive/My Drive/Colab Notebooks/UWL/ML_L6/'
    #import sys
    #sys.path.append('/content/gdrive/My Drive/Colab Notebooks')
    #%cd /content/gdrive/My Drive/Colab Notebooks
    #!ls
```

```
Drive already mounted at /content/gdrive; to attempt to forcibly remount,
```

```
[ ] import numpy as np
    import matplotlib.pyplot as plt
    import os
    import pickle

    plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
    plt.rcParams['image.interpolation'] = 'nearest'
    plt.rcParams['image.cmap'] = 'gray'
```

```
[ ] def load_CIFAR_batch(filename):
      # load single batch of cifar
      print(filename)
      with open(filename, 'rb') as f:
        datadict = pickle.load(f,encoding='latin1')
        X = datadict['data']
        Y = datadict['labels']
        X = X.reshape(10000, 3, 32, 32).transpose(0,2,3,1).astype("float")
        Y = np.array(Y)
        return X, Y
```

```
[ ] def load_CIFAR10(ROOT):
      # load all of cifar
      xs = []
      ys = []
      for b in range(1,6):
        f = os.path.join(ROOT, 'data_batch_%d' % (b, ))
        X, Y = load_CIFAR_batch(f)
        xs.append(X)
        ys.append(Y)
      Xtr = np.concatenate(xs)
      Ytr = np.concatenate(ys)
      del X, Y
      Xte, Yte = load_CIFAR_batch(os.path.join(ROOT, 'test_batch'))
      return Xtr, Ytr, Xte, Yte
```

```
[ ] X_train, y_train, X_test, y_test = load_CIFAR10('/content/gdrive/My Drive/Colab Notebooks/UWL/CIFR10/')
```

```
/content/gdrive/My Drive/Colab Notebooks/UWL/CIFR10/data_batch_1
/content/gdrive/My Drive/Colab Notebooks/UWL/CIFR10/data_batch_2
/content/gdrive/My Drive/Colab Notebooks/UWL/CIFR10/data_batch_3
/content/gdrive/My Drive/Colab Notebooks/UWL/CIFR10/data_batch_4
/content/gdrive/My Drive/Colab Notebooks/UWL/CIFR10/data_batch_5
/content/gdrive/My Drive/Colab Notebooks/UWL/CIFR10/test_batch
```

```
[ ] num_training = 5000
    mask = list(range(num_training))
    X_train = X_train[mask]
    y_train = y_train[mask]

    num_test = 500
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]
```

```
[ ] X_train.shape
```

```
(5000, 32, 32, 3)
```

```
[ ] X_train = np.reshape(X_train, (X_train.shape[0], -1))
    X_test = np.reshape(X_test, (X_test.shape[0], -1))
```

```
[ ] print(X_train.shape, X_test.shape)
```

```
(5000, 3072) (500, 3072)
```

```
[ ] class KNearestNeighbor(object):
      # a kNN classifier with L2 distance

      def __init__(self):
        pass

      def train(self, X, y):
        """
        Train the classifier. For k-nearest neighbors this is just
        memorizing the training data.
        Inputs:
        - X: A numpy array of shape (num_train, D) containing the training data
          consisting of num_train samples each of dimension D.
        - y: A numpy array of shape (N,) containing the training labels, where
             y[i] is the label for X[i].
        """
        self.X_train = X
        self.y_train = y

      def predict(self, X, k=1, num_loops=0):
        """
        Predict labels for test data using this classifier.
        Inputs:
        - X: A numpy array of shape (num_test, D) containing test data consisting
             of num_test samples each of dimension D.
        - k: The number of nearest neighbors that vote for the predicted labels.
        - num_loops: Determines which implementation to use to compute distances
          between training points and testing points.
        Returns:
        - y: A numpy array of shape (num_test,) containing predicted labels for the
          test data, where y[i] is the predicted label for the test point X[i].
        """
        if num_loops == 0:
          dists = self.compute_distances_no_loops(X)
        elif num_loops == 1:
          dists = self.compute_distances_one_loop(X)
        elif num_loops == 2:
          dists = self.compute_distances_two_loops(X)
        else:
          raise ValueError('Invalid value %d for num_loops' % num_loops)
```

```
[ ]     return self.predict_labels(dists, k=k)

      def compute_distances_two_loops(self, X):
        """
        Compute the distance between each test point in X and each training point
        in self.X_train using a nested loop over both the training data and the
        test data.
        Inputs:
        - X: A numpy array of shape (num_test, D) containing test data.
        Returns:
        - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
          is the Euclidean distance between the ith test point and the jth training
          point.
        """
        num_test = X.shape[0]
        num_train = self.X_train.shape[0]
        dists = np.zeros((num_test, num_train))
        for i in range(num_test):
          for j in range(num_train):

            # Compute the l2 distance between the ith test point and the jth
            # training point, and store the result in dists[i, j]. You should
            # not use a loop over dimension.

            dists[i, j] = np.sqrt(np.sum((X[i, :] - self.X_train[j, :]) **2))

        return dists

      def compute_distances_one_loop(self, X):
        """
        Compute the distance between each test point in X and each training point
        in self.X_train using a single loop over the test data.
        Input / Output: Same as compute_distances_two_loops
        """
        num_test = X.shape[0]
        num_train = self.X_train.shape[0]
        dists = np.zeros((num_test, num_train))
        for i in range(num_test):

          # Compute the l2 distance between the ith test point and all training
          # points, and store the result in dists[i, :].
```

```
[ ]        dists[i, :] = np.sqrt(np.sum((self.X_train - X[i, :])**2, axis=1))

        return dists

      def compute_distances_no_loops(self, X):
        """
        Compute the distance between each test point in X and each training point
        in self.X_train using no explicit loops.
        Input / Output: Same as compute_distances_two_loops
        """
        num_test = X.shape[0]
        num_train = self.X_train.shape[0]
        dists = np.zeros((num_test, num_train))

        # Compute the l2 distance between all test points and all training
        # points without using any explicit loops, and store the result in dists.

        # You should implement this function using only basic array operations;
        # in particular you should not use functions from scipy.
        # HINT: Try to formulate the l2 distance using matrix multiplication and two broadcast sums.

        dists = np.sqrt((-2 * np.dot(X, self.X_train.T)) + np.sum(X**2, axis=1, keepdims=True) + np.sum(self.X_train**2, axis=1))

        return dists

      def predict_labels(self, dists, k=1):
        """
        Given a matrix of distances between test points and training points,
        predict a label for each test point.
        Inputs:
        - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
          gives the distance betwen the ith test point and the jth training point.
        Returns:
        - y: A numpy array of shape (num_test,) containing predicted labels for the
          test data, where y[i] is the predicted label for the test point X[i].
        """
        num_test = dists.shape[0]
        y_pred = np.zeros(num_test)
        for i in range(num_test):
          # A list of length k storing the labels of the k nearest neighbors to
          # the ith test point.
```

```
closest_y = []

# Use the distance matrix to find the k nearest neighbors of the ith
# testing point, and use self.y_train to find the labels of these
# neighbors. Store these labels in closest_y.
# Hint: Look up the function numpy.argsort.

k_nearest_idxs = np.argsort(dists[i, :])[:k]
closest_y = self.y_train[k_nearest_idxs]

# Now that you have found the labels of the k nearest neighbors, you
# need to find the most common label in the list closest_y of labels.
# Store this label in y_pred[i]. Break ties by choosing the smaller
# label.

y_pred[i] = np.argmax(np.bincount(closest_y))

return y_pred
```

We use k = 1 (which is Nearest Neighbor).

```
[ ] y_test_pred = classifier.predict_labels(dists, k=1)
```

Compute and print the fraction of correctly predicted examples

```
[ ] num_correct = np.sum(y_test_pred == y_test)
    accuracy = float(num_correct) / num_test
    print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

```
Got 137 / 500 correct => accuracy: 0.274000
```

You should expect to see approximately 26% accuracy. Now lets try out a larger k, say k = 5:

```
[ ] y_test_pred = classifier.predict_labels(dists, k=5)
    num_correct = np.sum(y_test_pred == y_test)
    accuracy = float(num_correct) / num_test
    print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

```
Got 139 / 500 correct => accuracy: 0.278000
```

You should expect to see a slightly better performance than with `k = 1`.

Now lets speed up distance matrix computation by using partial vectorization with one loop.

Let us run the function compute_distances_one_loop and run the code below:

```
[ ] dists_one = classifier.compute_distances_one_loop(X_test)
```

We have implemented the k-Nearest Neighbor classifier but we set the value k = 5 arbitrarily.

We will now determine the best value of this hyperparameter with cross-validation.

Split up the training data into folds.

After splitting, X_train_folds and y_train_folds should each be lists of length num_folds, where X_train_folds[i].

Hint: Look up the numpy array_split function.

```
[ ] num_folds = 5
    k_choices = [1, 3, 5, 7, 9, 11, 13, 15]

    X_train_folds = []
    y_train_folds = []

    X_train_folds = np.split(X_train, num_folds)
    y_train_folds = np.split(y_train, num_folds)
```

```
[ ] k_to_accuracies = {}
    num_split = X_train.shape[0] / num_folds
    # Replace np.float with float or np.float64
    acc_k = np.zeros((len(k_choices), num_folds), dtype=float)
```

```
[ ] for ik ,k in enumerate(k_choices):
        for i in range(num_folds):
            train_set = np.concatenate((X_train_folds[:i]+X_train_folds[i+1:]))
            label_set = np.concatenate((y_train_folds[:i]+y_train_folds[i+1:]))
            classifier.train(train_set, label_set)
            y_pred_fold = classifier.predict(X_train_folds[i], k=k, num_loops=0)
            num_correct = np.sum(y_pred_fold == y_train_folds[i])
            acc_k[ik, i] = float(num_correct) / num_split
        k_to_accuracies[k] = acc_k[ik]
```

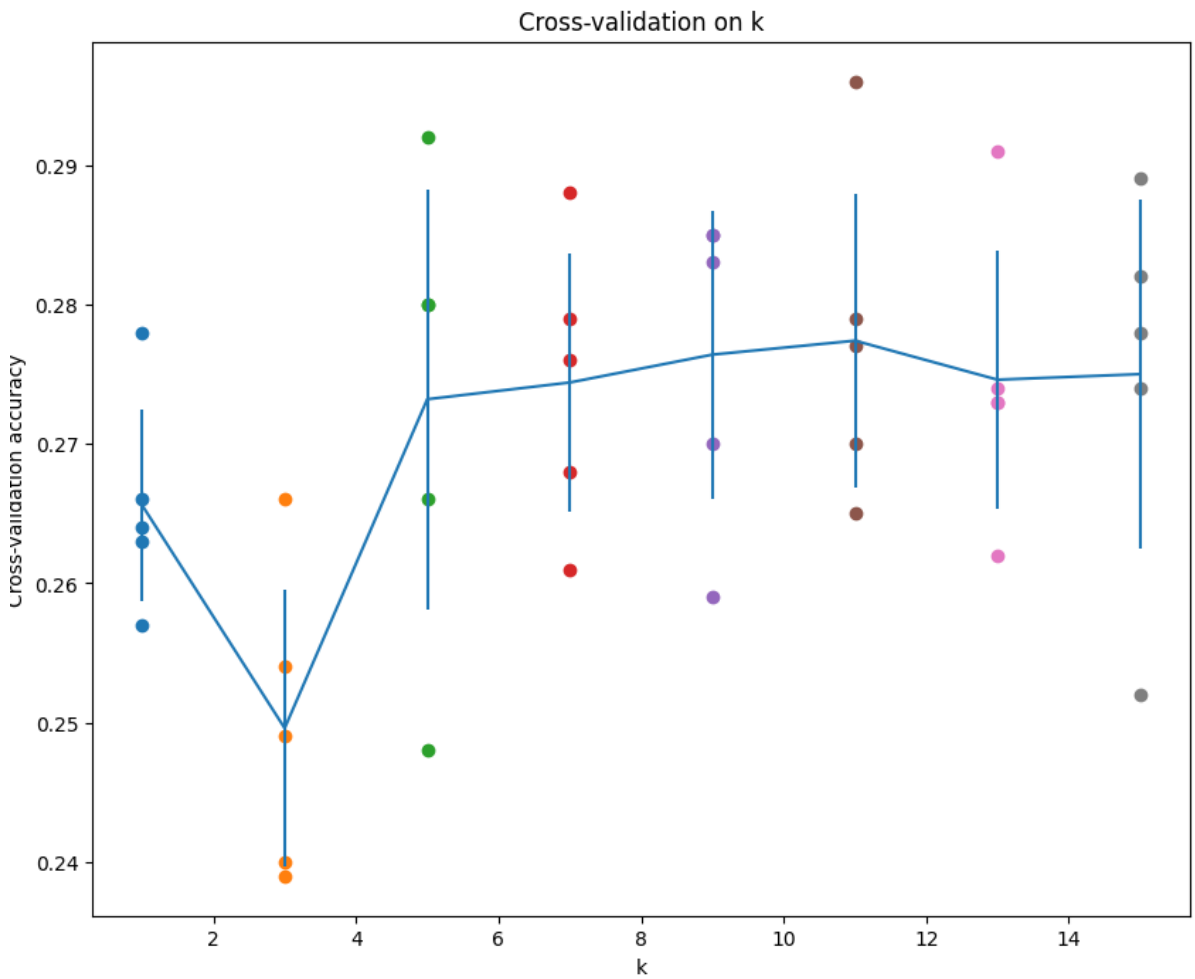
Print out the computed accuracies

```
[ ] for k in sorted(k_to_accuracies):
        for accuracy in k_to_accuracies[k]:
            print('k = %d, accuracy = %f' % (k, accuracy))
```

```
k = 1, accuracy = 0.263000
k = 1, accuracy = 0.257000
k = 1, accuracy = 0.264000
k = 1, accuracy = 0.278000
k = 1, accuracy = 0.266000
k = 3, accuracy = 0.239000
k = 3, accuracy = 0.249000
k = 3, accuracy = 0.240000
k = 3, accuracy = 0.266000
k = 3, accuracy = 0.254000
k = 5, accuracy = 0.248000
k = 5, accuracy = 0.266000
k = 5, accuracy = 0.280000
k = 5, accuracy = 0.292000
k = 5, accuracy = 0.280000
k = 7, accuracy = 0.261000
k = 7, accuracy = 0.279000
k = 7, accuracy = 0.268000
k = 7, accuracy = 0.288000
k = 7, accuracy = 0.276000
k = 9, accuracy = 0.259000
k = 9, accuracy = 0.283000
k = 9, accuracy = 0.270000
k = 9, accuracy = 0.285000
k = 9, accuracy = 0.285000
k = 11, accuracy = 0.265000
k = 11, accuracy = 0.296000
k = 11, accuracy = 0.277000
k = 11, accuracy = 0.279000
k = 11, accuracy = 0.270000
k = 13, accuracy = 0.262000
k = 13, accuracy = 0.291000
k = 13, accuracy = 0.273000
k = 13, accuracy = 0.274000
k = 13, accuracy = 0.273000
k = 15, accuracy = 0.252000
k = 15, accuracy = 0.289000
k = 15, accuracy = 0.278000
k = 15, accuracy = 0.282000
k = 15, accuracy = 0.274000
```

Plot the raw observations

```
[ ] for k in k_choices:
        accuracies = k_to_accuracies[k]
        plt.scatter([k] * len(accuracies), accuracies)

    # plot the trend line with error bars that correspond to standard deviation
    accuracies_mean = np.array([np.mean(v) for k,v in sorted(k_to_accuracies.items())])
    accuracies_std = np.array([np.std(v) for k,v in sorted(k_to_accuracies.items())])
    plt.errorbar(k_choices, accuracies_mean, yerr=accuracies_std)
    plt.title('Cross-validation on k')
    plt.xlabel('k')
    plt.ylabel('Cross-validation accuracy')
    plt.show()
```



```python
# Step 1: Choose the best k based on the cross-validation results
# From the cross-validation, choose the best k based on the mean accuracy
best_k = k_choices[np.argmax(accuracies_mean)]
print(f"The best value of k is: {best_k}")

# Step 2: Retrain the classifier with half of the CIFAR-10 training data (2500 samples)
half_X_train = X_train[:2500]
half_y_train = y_train[:2500]

# Initialize and train the k-NN classifier
classifier = KNearestNeighbor()
classifier.train(half_X_train, half_y_train)

# Step 3: Test the model on the full test data
y_test_pred = classifier.predict(X_test, k=best_k, num_loops=0)

# Step 1: Train on halved data
half_X_train = X_train[:2500]
half_y_train = y_train[:2500]

classifier_half = KNearestNeighbor()
classifier_half.train(half_X_train, half_y_train)

# Test on the test set using halved training data
y_test_pred_half = classifier_half.predict(X_test, k=best_k)
test_accuracy_half = np.mean(y_test_pred_half == y_test)
print(f"Test accuracy with halved training data and k = {best_k}: {test_accuracy_half * 100:.2f}%")

# Step 2: Train on full data
classifier_full = KNearestNeighbor()
classifier_full.train(X_train, y_train)

# Test on the test set using full training data
y_test_pred_full = classifier_full.predict(X_test, k=best_k)
test_accuracy_full = np.mean(y_test_pred_full == y_test)
print(f"Test accuracy with full training data and k = {best_k}: {test_accuracy_full * 100:.2f}%")
```

```
The best value of k is: 11
Test accuracy with halved training data and k = 11: 24.60%
Test accuracy with full training data and k = 11: 26.20%
```

Based on Cross Validation, we can easily say that the best k values **is k – 11** as it provided **the highest average accuracy across** multiple validation runs.

Also while calculating mathematically the average accuracy of k -11 is also:

$$k\ 11 = \frac{(0.265+0.296+0.277+0.279+0.270)}{5} = \underline{0.2774}$$

As you can see from the Screenshots of the output, the accuracies are:

Accuracy with halved training data = 24.60 %

Accuracy with full training data = 26.20 %

What this means?

With fewer training examples, the classifier may not have enough data to learn the true distribution of the classes. This could lead to a decrease in test accuracy, as the algorithm might struggle to make correct predictions for some test points.

Week – 10 – Classification (Convolutional Neural Network)

The core operation in Convolutional Networks is the **convolution**. A **small filter (or kernel)** slides over the input data, performing an element-wise multiplication, and sums the results to produce a **feature map**, this detects spatial patterns such as edges, textures, or shapes in images.

CNN work by **moving small filters across** the input image. This means the filters are re-used for recognizing patterns throughout the entire input image. This makes the Convolutional Networks much more powerful than Fully-Connected networks with the same number of variables. This in turn makes the Convolutional Networks faster to train.

We can also take convolutions of convolutions -- we can stack as many convolutions as we want, as long as there are enough pixels to fit a kernel.

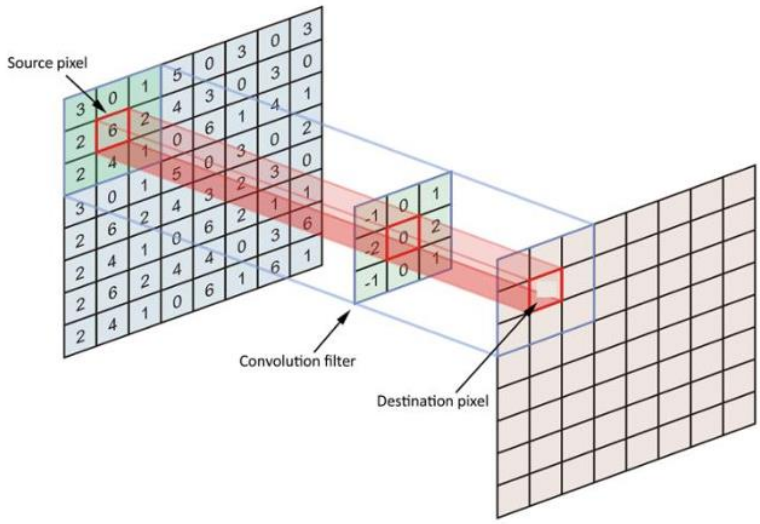
The convolutional filters are initially chosen at random, so the classification is done randomly. The error between the predicted and true class of the input image is measured as the so-called **cross-entropy**. The optimizer then automatically propagates this error back through the Convolutional Network using **the chain-rule of differentiation** and updates the filter-weights so as to improve the classification error. This is done iteratively thousands of times until the classification error is sufficiently low.

Collab Links

https://colab.research.google.com/drive/1KdW_ZY4qbVCqvfDzsJC2UiFJDfkyg2Bg?usp=sharing

Task 1

For the convolution operation given below, compute the destination pixel value in the output image by applying the 3×3 filter to the source (input) image. Provide the details of your calculations.



Source Pixel = $\begin{bmatrix} 3 & 0 & 1 \\ 2 & 6 & 2 \\ 2 & 4 & 1 \end{bmatrix}$ Convolution Filter = $\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$

Cross multiplying the rows in Cross Pixels to itself we get:

Row 1 = (3 x -1) + (0 x 0) + (1 x 1)

Row 2 = (2 x -2) + (6 x 0) + (2 x 2)

Row 3 = (2 x -1) + (4 x 0) + (1 x 1)

And adding them all together we get:

Destination Pixel = Row 1 + Row 2 + Row 3

Destination Pixel = [(3 x -1) + (0 x 0) + (1 x 1)] + [(2 x -2) + (6 x 0) + (2 x 2)] +

[(2 x -1) + (4 x 0) + (1 x 1)]

= **-3**

Hence the **Destination Pixel is found to be -3**

Task 2

Do the following:

- Train the your CNN model for a few more iterations (you can do this by changing the number of **epochs** in **model.fit**, then click on **Runtime** and **Run all**)
- Try it for 10, 20, 50 epochs.
- How does the acuuracy change? Discuss your observations.
- If accuracy is improving, explain why. If accuracy is getting worse, why do you think that is?
- Change the kernel size from (3, 3) to (5, 5) in your CNN model (for all layers.Conv2D layers)
- Would that help to improve the accuracy?

Ans:

10 epochs kernel (3,3)

Import TensorFlow

```
[ ] import tensorflow as tf
    from tensorflow.keras import datasets, layers, models
    import matplotlib.pyplot as plt
```

Download and prepare the CIFAR10 dataset

The CIFAR10 dataset contains 60,000 color images in 10 classes, with 6,000 images in each class. The dataset is divided into 50,000 training images and 10,000 testing images. The classes are mutually exclusive and there is no overlap between them.

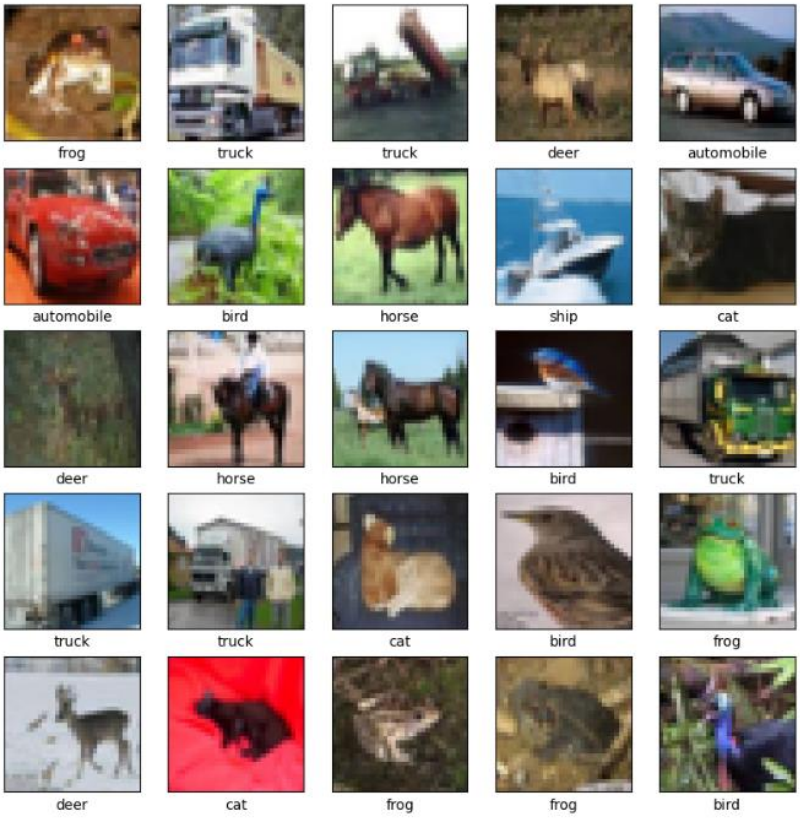
```
[ ] (train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()

    # Normalize pixel values to be between 0 and 1
    train_images, test_images = train_images / 255.0, test_images / 255.0
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 ━━━━━━━━━━━━━━━━━━━━ 4s 0us/step
```

Verify the data

To verify that the dataset looks correct, let's plot the first 25 images from the training set and display the class name below each image.

```
[3] class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
                   'dog', 'frog', 'horse', 'ship', 'truck']

    plt.figure(figsize=(10,10))
    for i in range(25):
        plt.subplot(5,5,i+1)
        plt.xticks([])
        plt.yticks([])
        plt.grid(False)
        plt.imshow(train_images[i], cmap=plt.cm.binary)
        # The CIFAR labels happen to be arrays,
        # which is why you need the extra index
        plt.xlabel(class_names[train_labels[i][0]])
    plt.show()
```



Create the convolutional base

```
[ ] model = models.Sequential()
    model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(64, (3, 3), activation='relu'))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not pass an
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Let's display the architecture of our model so far.

```
[ ] model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d (MaxPooling2D)	(None, 15, 15, 32)	0
conv2d_1 (Conv2D)	(None, 13, 13, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 6, 6, 64)	0
conv2d_2 (Conv2D)	(None, 4, 4, 64)	36,928

Total params: 56,320 (220.00 KB)
Trainable params: 56,320 (220.00 KB)
Non-trainable params: 0 (0.00 B)

```
[ ] model.add(layers.Flatten())
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(10))
```

Here's the complete architecture of our model.

```
[ ] model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d (MaxPooling2D)	(None, 15, 15, 32)	0
conv2d_1 (Conv2D)	(None, 13, 13, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 6, 6, 64)	0
conv2d_2 (Conv2D)	(None, 4, 4, 64)	36,928
flatten (Flatten)	(None, 1024)	0
dense (Dense)	(None, 64)	65,600
dense_1 (Dense)	(None, 10)	650

Total params: 122,570 (478.79 KB)
Trainable params: 122,570 (478.79 KB)
Non-trainable params: 0 (0.00 B)

As you can see, our (4, 4, 64) outputs were flattened into vectors of shape (1024) before going through two Dense layers.

Compile and train the model

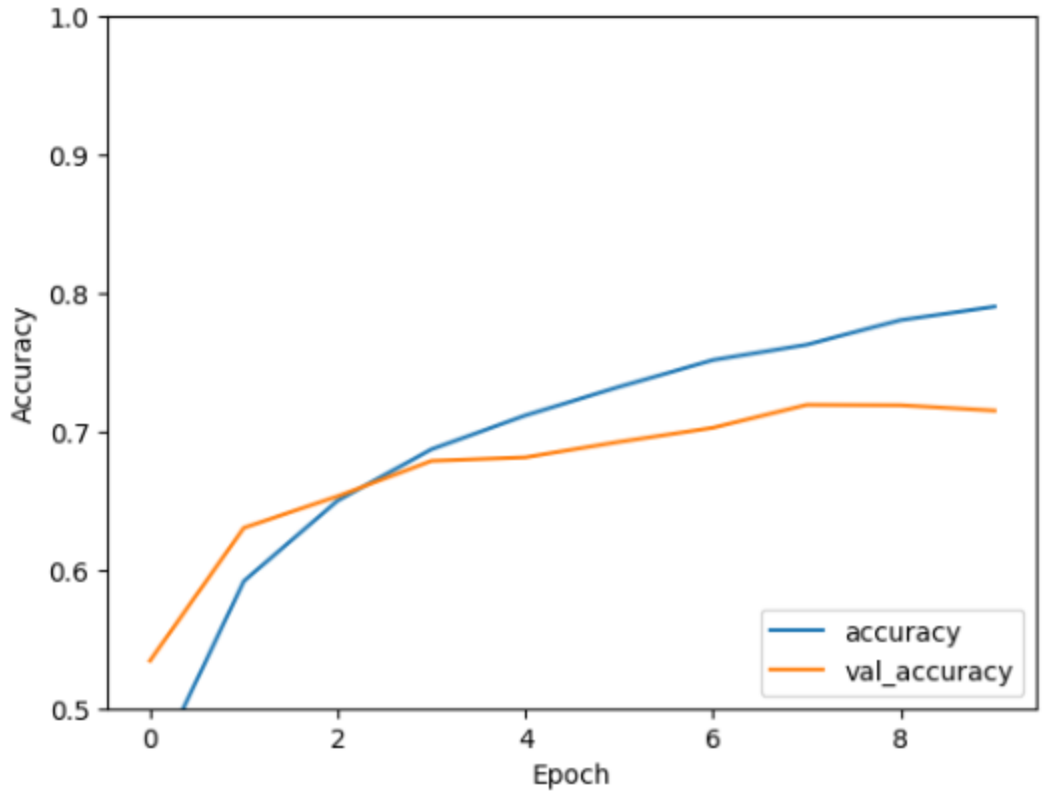
```
[ ] model.compile(optimizer='adam',
                  loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                  metrics=['accuracy'])

    history = model.fit(train_images, train_labels, epochs=10,
                        validation_data=(test_images, test_labels))
```

```
Epoch 1/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 13s 5ms/step - accuracy: 0.3537 - loss: 1.7583 - val_accuracy: 0.5343 - val_loss: 1.28
Epoch 2/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 14s 3ms/step - accuracy: 0.5783 - loss: 1.1891 - val_accuracy: 0.6304 - val_loss: 1.04
Epoch 3/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.6444 - loss: 1.0072 - val_accuracy: 0.6532 - val_loss: 0.988
Epoch 4/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.6869 - loss: 0.8939 - val_accuracy: 0.6788 - val_loss: 0.924
Epoch 5/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.7130 - loss: 0.8145 - val_accuracy: 0.6813 - val_loss: 0.920
Epoch 6/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.7362 - loss: 0.7525 - val_accuracy: 0.6925 - val_loss: 0.889
Epoch 7/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.7531 - loss: 0.7048 - val_accuracy: 0.7027 - val_loss: 0.878
Epoch 8/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.7645 - loss: 0.6651 - val_accuracy: 0.7193 - val_loss: 0.847
Epoch 9/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - accuracy: 0.7855 - loss: 0.6135 - val_accuracy: 0.7190 - val_loss: 0.830
Epoch 10/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.7953 - loss: 0.5795 - val_accuracy: 0.7151 - val_loss: 0.853
```

Evaluate the model

```
[ ] plt.plot(history.history['accuracy'], label='accuracy')
    plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.ylim([0.5, 1])
    plt.legend(loc='lower right')
```

```
<matplotlib.legend.Legend at 0x7a6f39b74220>
```




```
[ ] test_loss, test_acc = model.evaluate(test_images,  test_labels)
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.7176 - loss: 0.8510
```

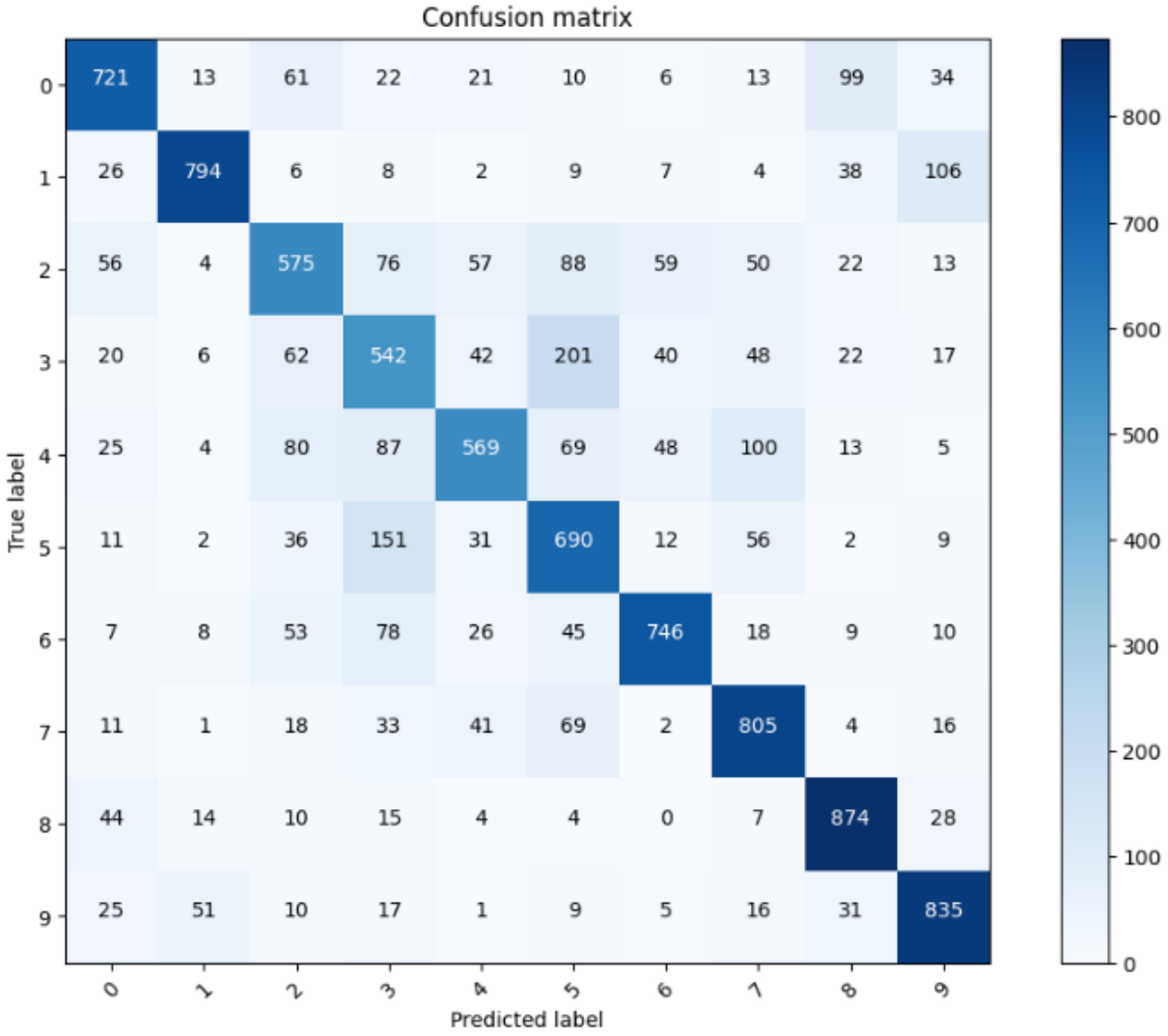
```
[ ] print('Test accuracy:', test_acc)
```

```
Test accuracy: 0.7150999903678894
```

Plot confusion matrix

```
[ ] from sklearn.metrics import confusion_matrix
    import itertools
    import numpy as np
    plt.rcParams['figure.figsize'] = [10,7]

    def plot_confusion_matrix(cm, classes,
                              normalize=False,
                              title='Confusion matrix',
                              cmap=plt.cm.Blues):

      if normalize:
          cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
          print("Normalized confusion matrix")
      else:
          print('Confusion matrix, without normalization')

      print(cm)

      plt.imshow(cm, interpolation='nearest', cmap=cmap)
      plt.title(title)
      plt.colorbar()
      tick_marks = np.arange(len(classes))
      plt.xticks(tick_marks, classes, rotation=45)
      plt.yticks(tick_marks, classes)

      fmt = '.2f' if normalize else 'd'
      thresh = cm.max() / 2.
      for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
          plt.text(j, i, format(cm[i, j], fmt),
                   horizontalalignment="center",
                   color="white" if cm[i, j] > thresh else "black")

      plt.tight_layout()
      plt.ylabel('True label')
      plt.xlabel('Predicted label')
      plt.show()


    predicted_labels = model.predict(test_images).argmax(axis=1)
    cm = confusion_matrix(test_labels, predicted_labels)
    plot_confusion_matrix(cm, list(range(10)))
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
Confusion matrix, without normalization
[[721  13  61  22  21  10   6  13  99  34]
 [ 26 794   6   8   2   9   7   4  38 106]
 [ 56   4 575  76  57  88  59  50  22  13]
 [ 20   6  62 542  42 201  40  48  22  17]
 [ 25   4  80  87 569  69  48 100  13   5]
 [ 11   2  36 151  31 690  12  56   2   9]
 [  7   8  53  78  26  45 746  18   9  10]
 [ 11   1  18  33  41  69   2 805   4  16]
 [ 44  14  10  15   4   4   0   7 874  28]
 [ 25  51  10  17   1   9   5  16  31 835]]
```



Collab Link:
https://colab.research.google.com/drive/1OketYrykAc0dajiy8YUQSABEtxgybarX?usp=sharing

10 epochs kernel (5,5)

Create the convolutional base

```
[ ] model = models.Sequential()
    model.add(layers.Conv2D(32, (5, 5), activation='relu', input_shape=(32, 32, 3)))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(64, (5, 5), activation='relu'))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(64, (5, 5), activation='relu'))
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not pass an `input_shape`/`ir
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Let's display the architecture of our model so far.

```
[ ] model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 32)	2,432
max_pooling2d (MaxPooling2D)	(None, 14, 14, 32)	0
conv2d_1 (Conv2D)	(None, 10, 10, 64)	51,264
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
conv2d_2 (Conv2D)	(None, 1, 1, 64)	102,464

Total params: 156,160 (610.00 KB)
Trainable params: 156,160 (610.00 KB)
Non-trainable params: 0 (0.00 B)

```
[ ] model.add(layers.Flatten())
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(10))
```

Here's the complete architecture of our model.

```
[ ] model.summary()
```

```
[ ] Model: "sequential"
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 32)	2,432
max_pooling2d (MaxPooling2D)	(None, 14, 14, 32)	0
conv2d_1 (Conv2D)	(None, 10, 10, 64)	51,264
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
conv2d_2 (Conv2D)	(None, 1, 1, 64)	102,464
flatten (Flatten)	(None, 64)	0
dense (Dense)	(None, 64)	4,160
dense_1 (Dense)	(None, 10)	650

```
Total params: 160,970 (628.79 KB)
Trainable params: 160,970 (628.79 KB)
Non-trainable params: 0 (0.00 B)
```

As you can see, our (4, 4, 64) outputs were flattened into vectors of shape (1024) before going through two Dense layers.

Compile and train the model

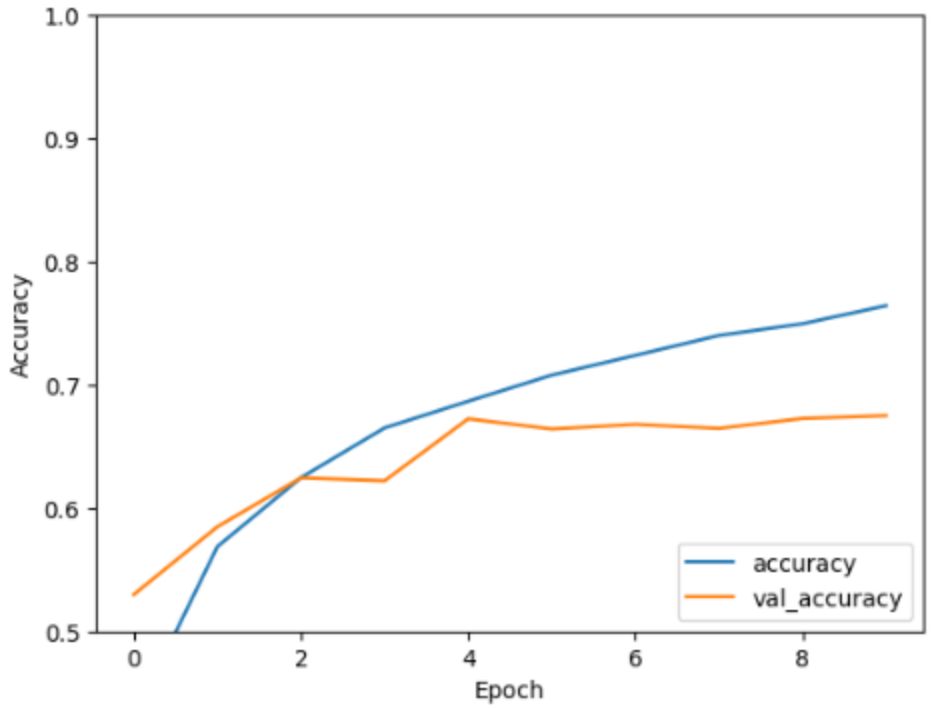
```
[ ] model.compile(optimizer='adam',
                  loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                  metrics=['accuracy'])

    history = model.fit(train_images, train_labels, epochs=10,
                        validation_data=(test_images, test_labels))
```

```
Epoch 1/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 101s 63ms/step - accuracy: 0.3392 - loss: 1.7775 - val_accuracy: 0.5298 - val_loss: 1.3158
Epoch 2/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 98s 63ms/step - accuracy: 0.5510 - loss: 1.2650 - val_accuracy: 0.5848 - val_loss: 1.1977
Epoch 3/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 142s 63ms/step - accuracy: 0.6169 - loss: 1.0852 - val_accuracy: 0.6245 - val_loss: 1.0633
Epoch 4/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 97s 62ms/step - accuracy: 0.6613 - loss: 0.9606 - val_accuracy: 0.6221 - val_loss: 1.0976
Epoch 5/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 147s 66ms/step - accuracy: 0.6895 - loss: 0.8785 - val_accuracy: 0.6724 - val_loss: 0.9598
Epoch 6/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 138s 63ms/step - accuracy: 0.7126 - loss: 0.8199 - val_accuracy: 0.6641 - val_loss: 0.9715
Epoch 7/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 141s 63ms/step - accuracy: 0.7283 - loss: 0.7706 - val_accuracy: 0.6679 - val_loss: 0.9606
Epoch 8/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 144s 64ms/step - accuracy: 0.7430 - loss: 0.7259 - val_accuracy: 0.6646 - val_loss: 0.9869
Epoch 9/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 141s 64ms/step - accuracy: 0.7557 - loss: 0.6824 - val_accuracy: 0.6727 - val_loss: 0.9734
Epoch 10/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 143s 65ms/step - accuracy: 0.7713 - loss: 0.6479 - val_accuracy: 0.6750 - val_loss: 0.9872
```

Evaluate the model

```
[ ] plt.plot(history.history['accuracy'], label='accuracy')
    plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.ylim([0.5, 1])
    plt.legend(loc='lower right')
```

```
<matplotlib.legend.Legend at 0x7b43ce2fc0a0>
```




```
[ ] test_loss, test_acc = model.evaluate(test_images,  test_labels)
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 6s 20ms/step - accuracy: 0.6790 - loss: 0.9688
```

```
[ ] print('Test accuracy:', test_acc)
```

```
Test accuracy: 0.675000011920929
```

Collab Link:
https://colab.research.google.com/drive/1zSDj4tH1X4uLKmQxYispiO8BFelizJWG?usp=sharing

20 epochs kernel (3,3)

Create the convolutional base

```
[ ] model = models.Sequential()
    model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(64, (3, 3), activation='relu'))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Let's display the architecture of our model so far.

```
[ ] model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d (MaxPooling2D)	(None, 15, 15, 32)	0
conv2d_1 (Conv2D)	(None, 13, 13, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 6, 6, 64)	0
conv2d_2 (Conv2D)	(None, 4, 4, 64)	36,928

Total params: 56,320 (220.00 KB)
Trainable params: 56,320 (220.00 KB)
Non-trainable params: 0 (0.00 B)

```
[ ] model.add(layers.Flatten())
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(10))
```

Here's the complete architecture of our model.

```
[ ] model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d (MaxPooling2D)	(None, 15, 15, 32)	0
conv2d_1 (Conv2D)	(None, 13, 13, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 6, 6, 64)	0
conv2d_2 (Conv2D)	(None, 4, 4, 64)	36,928
flatten (Flatten)	(None, 1024)	0
dense (Dense)	(None, 64)	65,600
dense_1 (Dense)	(None, 10)	650

Total params: 122,570 (478.79 KB)
Trainable params: 122,570 (478.79 KB)
Non-trainable params: 0 (0.00 B)

As you can see, our (4, 4, 64) outputs were flattened into vectors of shape (1024) before going through two Dense layers.

Compile and train the model

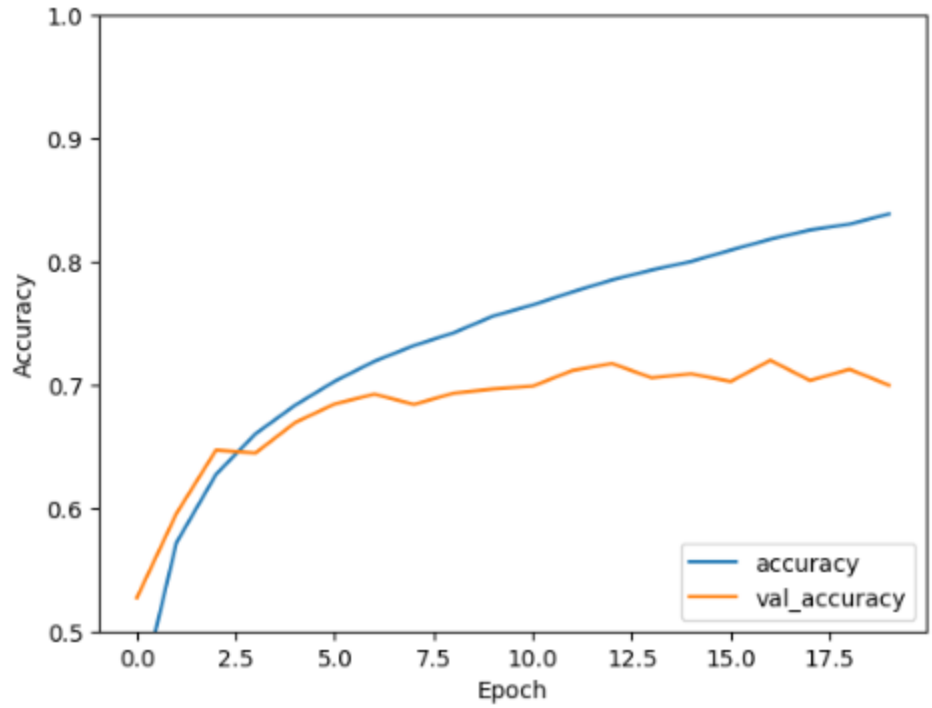
```
[ ] model.compile(optimizer='adam',
                  loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                  metrics=['accuracy'])

    history = model.fit(train_images, train_labels, epochs=20,
                        validation_data=(test_images, test_labels))
```

```
Epoch 1/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 12s 4ms/step - accuracy: 0.3400 - loss: 1.7745 - val_accuracy: 0.5271 - val_loss: 1.3265
Epoch 2/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.5589 - loss: 1.2353 - val_accuracy: 0.5955 - val_loss: 1.1140
Epoch 3/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.6221 - loss: 1.0716 - val_accuracy: 0.6470 - val_loss: 1.0147
Epoch 4/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.6559 - loss: 0.9767 - val_accuracy: 0.6447 - val_loss: 1.0026
Epoch 5/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.6818 - loss: 0.9030 - val_accuracy: 0.6694 - val_loss: 0.9456
Epoch 6/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.7014 - loss: 0.8513 - val_accuracy: 0.6845 - val_loss: 0.9111
Epoch 7/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.7168 - loss: 0.7985 - val_accuracy: 0.6924 - val_loss: 0.8876
Epoch 8/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 10s 4ms/step - accuracy: 0.7387 - loss: 0.7444 - val_accuracy: 0.6841 - val_loss: 0.9074
Epoch 9/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.7463 - loss: 0.7191 - val_accuracy: 0.6932 - val_loss: 0.9017
Epoch 10/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.7578 - loss: 0.6838 - val_accuracy: 0.6966 - val_loss: 0.8997
Epoch 11/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.7662 - loss: 0.6602 - val_accuracy: 0.6989 - val_loss: 0.9181
Epoch 12/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 11s 4ms/step - accuracy: 0.7789 - loss: 0.6311 - val_accuracy: 0.7116 - val_loss: 0.8814
Epoch 13/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - accuracy: 0.7934 - loss: 0.5840 - val_accuracy: 0.7173 - val_loss: 0.8746
Epoch 14/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.8012 - loss: 0.5653 - val_accuracy: 0.7057 - val_loss: 0.8994
Epoch 15/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.8060 - loss: 0.5482 - val_accuracy: 0.7089 - val_loss: 0.8980
Epoch 16/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8177 - loss: 0.5180 - val_accuracy: 0.7027 - val_loss: 0.9533
Epoch 17/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.8265 - loss: 0.4914 - val_accuracy: 0.7199 - val_loss: 0.8943
Epoch 18/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.8343 - loss: 0.4730 - val_accuracy: 0.7035 - val_loss: 0.9645
Epoch 19/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.8395 - loss: 0.4599 - val_accuracy: 0.7126 - val_loss: 0.9713
Epoch 20/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.8465 - loss: 0.4375 - val_accuracy: 0.6996 - val_loss: 1.0498
```

Evaluate the model

```
[ ] plt.plot(history.history['accuracy'], label='accuracy')
    plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.ylim([0.5, 1])
    plt.legend(loc='lower right')
```

```
<matplotlib.legend.Legend at 0x7c4f63748b50>
```




```
[ ] test_loss, test_acc = model.evaluate(test_images,  test_labels)
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.7017 - loss: 1.0222
```

```
[ ] print('Test accuracy:', test_acc)
```

```
Test accuracy: 0.6995999813079834
```

Collab Link:
https://colab.research.google.com/drive/1OmyhAlKzU9nbDnXztGcWwLZAYdOst7SY?usp=sharing

20 epochs kernel (5,5)

Create the convolutional base

```
[ ] model = models.Sequential()
    model.add(layers.Conv2D(32, (5, 5), activation='relu', input_shape=(32, 32, 3)))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(64, (5, 5), activation='relu'))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(64, (5, 5), activation='relu'))
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not pass a
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Let's display the architecture of our model so far.

```
[ ] model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 32)	2,432
max_pooling2d (MaxPooling2D)	(None, 14, 14, 32)	0
conv2d_1 (Conv2D)	(None, 10, 10, 64)	51,264
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
conv2d_2 (Conv2D)	(None, 1, 1, 64)	102,464

Total params: 156,160 (610.00 KB)
Trainable params: 156,160 (610.00 KB)
Non-trainable params: 0 (0.00 B)

```
[ ] model.add(layers.Flatten())
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(10))
```

Here's the complete architecture of our model.

```
[ ] model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 32)	2,432
max_pooling2d (MaxPooling2D)	(None, 14, 14, 32)	0
conv2d_1 (Conv2D)	(None, 10, 10, 64)	51,264
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
conv2d_2 (Conv2D)	(None, 1, 1, 64)	102,464
flatten (Flatten)	(None, 64)	0
dense (Dense)	(None, 64)	4,160
dense_1 (Dense)	(None, 10)	650

Total params: 160,970 (628.79 KB)
Trainable params: 160,970 (628.79 KB)
Non-trainable params: 0 (0.00 B)

As you can see, our (4, 4, 64) outputs were flattened into vectors of shape (1024) before going through two Dense layers.

Compile and train the model

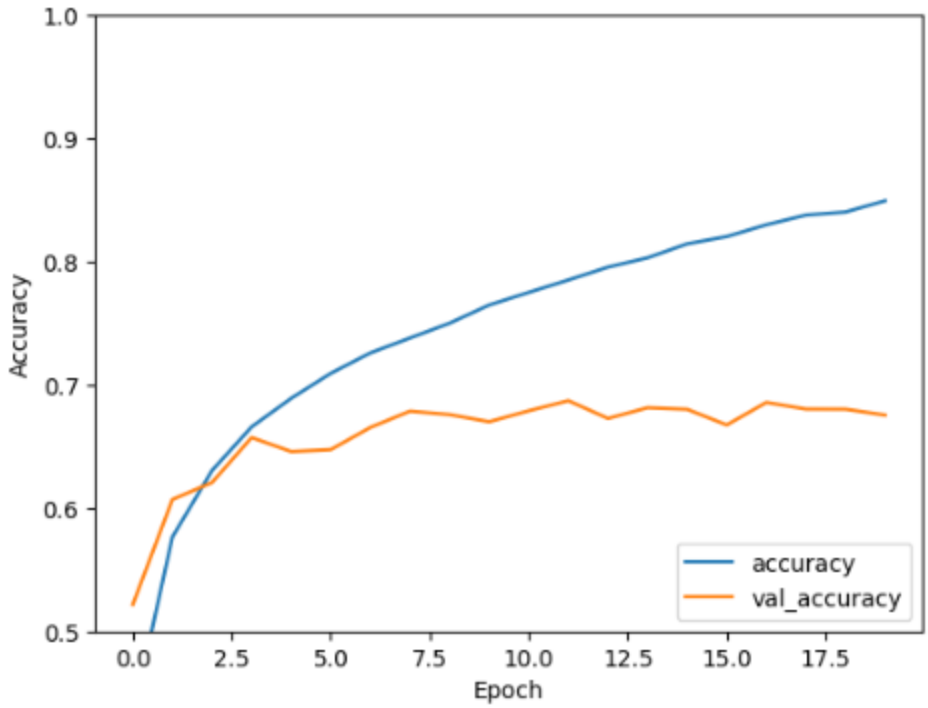
```
[ ] model.compile(optimizer='adam',
                  loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                  metrics=['accuracy'])

    history = model.fit(train_images, train_labels, epochs=20,
                        validation_data=(test_images, test_labels))
```

```
Epoch 1/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 12s 5ms/step - accuracy: 0.3378 - loss: 1.7800 - val_accuracy: 0.5217 - val_loss: 1.3262
Epoch 2/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.5609 - loss: 1.2282 - val_accuracy: 0.6069 - val_loss: 1.1126
Epoch 3/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.6263 - loss: 1.0638 - val_accuracy: 0.6207 - val_loss: 1.0902
Epoch 4/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.6661 - loss: 0.9568 - val_accuracy: 0.6572 - val_loss: 0.9864
Epoch 5/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.6903 - loss: 0.8814 - val_accuracy: 0.6458 - val_loss: 1.0390
Epoch 6/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.7129 - loss: 0.8200 - val_accuracy: 0.6474 - val_loss: 1.0296
Epoch 7/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.7265 - loss: 0.7727 - val_accuracy: 0.6654 - val_loss: 0.9919
Epoch 8/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - accuracy: 0.7414 - loss: 0.7298 - val_accuracy: 0.6786 - val_loss: 0.9495
Epoch 9/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.7604 - loss: 0.6766 - val_accuracy: 0.6758 - val_loss: 0.9917
Epoch 10/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.7708 - loss: 0.6517 - val_accuracy: 0.6700 - val_loss: 0.9989
Epoch 11/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.7809 - loss: 0.6180 - val_accuracy: 0.6789 - val_loss: 0.9723
Epoch 12/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.7935 - loss: 0.5912 - val_accuracy: 0.6871 - val_loss: 0.9857
Epoch 13/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.8007 - loss: 0.5648 - val_accuracy: 0.6727 - val_loss: 1.0259
Epoch 14/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.8110 - loss: 0.5358 - val_accuracy: 0.6814 - val_loss: 1.0289
Epoch 15/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.8202 - loss: 0.5105 - val_accuracy: 0.6801 - val_loss: 1.0431
Epoch 16/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8259 - loss: 0.4967 - val_accuracy: 0.6674 - val_loss: 1.1778
Epoch 17/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.8388 - loss: 0.4585 - val_accuracy: 0.6857 - val_loss: 1.1123
Epoch 18/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 11s 4ms/step - accuracy: 0.8455 - loss: 0.4433 - val_accuracy: 0.6803 - val_loss: 1.1481
Epoch 19/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.8482 - loss: 0.4286 - val_accuracy: 0.6802 - val_loss: 1.1433
Epoch 20/20
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.8582 - loss: 0.4013 - val_accuracy: 0.6754 - val_loss: 1.1948
```

Evaluate the model

```
[ ] plt.plot(history.history['accuracy'], label='accuracy')
    plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.ylim([0.5, 1])
    plt.legend(loc='lower right')
```

```
<matplotlib.legend.Legend at 0x7901d464a6e0>
```




```
[ ] test_loss, test_acc = model.evaluate(test_images,  test_labels)
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.6806 - loss: 1.1628
```

```
[ ] print('Test accuracy:', test_acc)
```

```
Test accuracy: 0.6754000186920166
```

Collab Link:

https://colab.research.google.com/drive/1v1AqE8sKLd4h7H1RS-vQUFayRdBdeNI0?usp=sharing

50 epochs kernel (3,3)

Create the convolutional base

```
[ ] model = models.Sequential()
    model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(64, (3, 3), activation='relu'))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not pass
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Let's display the architecture of our model so far.

```
[ ] model.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d_3 (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d_2 (MaxPooling2D)	(None, 15, 15, 32)	0
conv2d_4 (Conv2D)	(None, 13, 13, 64)	18,496
max_pooling2d_3 (MaxPooling2D)	(None, 6, 6, 64)	0
conv2d_5 (Conv2D)	(None, 4, 4, 64)	36,928

Total params: 56,320 (220.00 KB)
Trainable params: 56,320 (220.00 KB)
Non-trainable params: 0 (0.00 B)

```
[ ] model.add(layers.Flatten())
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(10))
```

Here's the complete architecture of our model.

```
[ ] model.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d_3 (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d_2 (MaxPooling2D)	(None, 15, 15, 32)	0
conv2d_4 (Conv2D)	(None, 13, 13, 64)	18,496
max_pooling2d_3 (MaxPooling2D)	(None, 6, 6, 64)	0
conv2d_5 (Conv2D)	(None, 4, 4, 64)	36,928
flatten_1 (Flatten)	(None, 1024)	0
dense_2 (Dense)	(None, 64)	65,600
dense_3 (Dense)	(None, 10)	650

Total params: 122,570 (478.79 KB)
Trainable params: 122,570 (478.79 KB)
Non-trainable params: 0 (0.00 B)

As you can see, our (4, 4, 64) outputs were flattened into vectors of shape (1024) before going through two Dense layers.

Compile and train the model

```
[ ] model.compile(optimizer='adam',
                  loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                  metrics=['accuracy'])

    history = model.fit(train_images, train_labels, epochs=50,
                        validation_data=(test_images, test_labels))
```

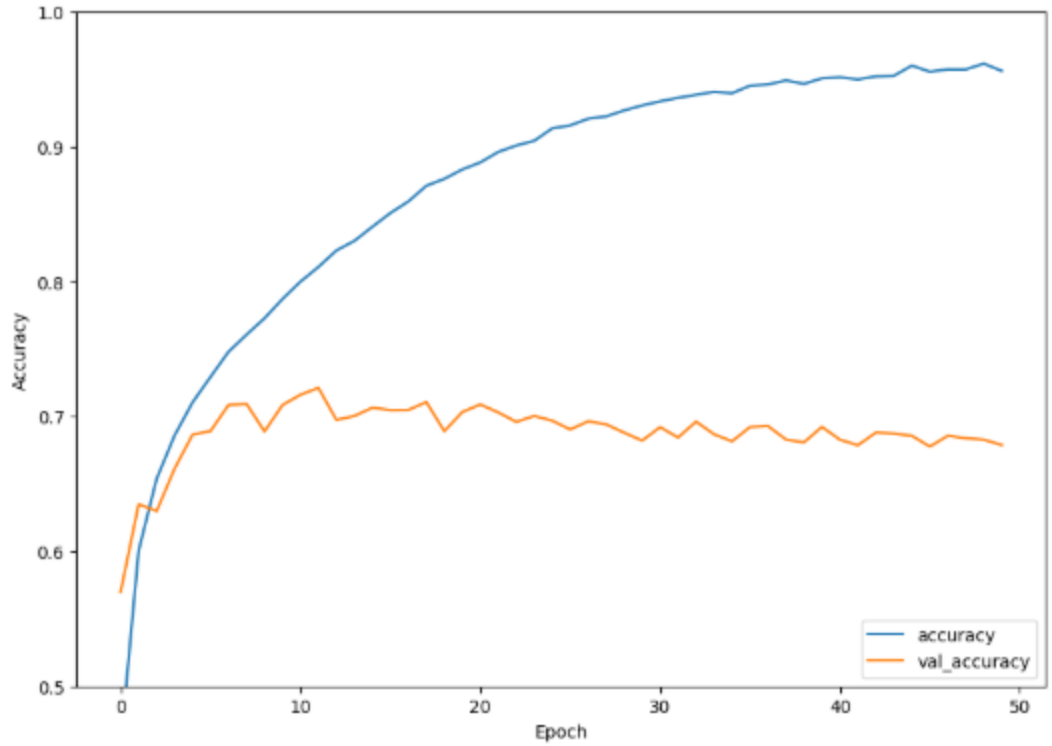
```
Epoch 1/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 84s 51ms/step - accuracy: 0.3612 - loss: 1.7313 - val_accuracy: 0.5700 - val_loss: 1.2084
Epoch 2/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 75s 48ms/step - accuracy: 0.5830 - loss: 1.1762 - val_accuracy: 0.6349 - val_loss: 1.0405
Epoch 3/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 83s 49ms/step - accuracy: 0.6450 - loss: 1.0020 - val_accuracy: 0.6298 - val_loss: 1.0590
Epoch 4/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 81s 48ms/step - accuracy: 0.6846 - loss: 0.8973 - val_accuracy: 0.6613 - val_loss: 0.9864
Epoch 5/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 76s 48ms/step - accuracy: 0.7130 - loss: 0.8213 - val_accuracy: 0.6865 - val_loss: 0.8945
Epoch 6/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 74s 47ms/step - accuracy: 0.7302 - loss: 0.7642 - val_accuracy: 0.6891 - val_loss: 0.8842
Epoch 7/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 82s 47ms/step - accuracy: 0.7522 - loss: 0.7093 - val_accuracy: 0.7085 - val_loss: 0.8566
Epoch 8/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 88s 51ms/step - accuracy: 0.7644 - loss: 0.6667 - val_accuracy: 0.7092 - val_loss: 0.8473
Epoch 9/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 75s 47ms/step - accuracy: 0.7784 - loss: 0.6238 - val_accuracy: 0.6888 - val_loss: 0.9406
Epoch 10/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 82s 47ms/step - accuracy: 0.7901 - loss: 0.5862 - val_accuracy: 0.7084 - val_loss: 0.8670
Epoch 11/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 82s 47ms/step - accuracy: 0.8059 - loss: 0.5501 - val_accuracy: 0.7160 - val_loss: 0.8859
Epoch 12/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 74s 47ms/step - accuracy: 0.8172 - loss: 0.5163 - val_accuracy: 0.7211 - val_loss: 0.8694
Epoch 13/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 82s 47ms/step - accuracy: 0.8317 - loss: 0.4821 - val_accuracy: 0.6973 - val_loss: 0.9243
Epoch 14/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 82s 47ms/step - accuracy: 0.8372 - loss: 0.4567 - val_accuracy: 0.7003 - val_loss: 1.0093
Epoch 15/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 82s 47ms/step - accuracy: 0.8498 - loss: 0.4229 - val_accuracy: 0.7064 - val_loss: 0.9711
Epoch 16/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 74s 47ms/step - accuracy: 0.8581 - loss: 0.3970 - val_accuracy: 0.7046 - val_loss: 1.0070
Epoch 17/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 82s 47ms/step - accuracy: 0.8682 - loss: 0.3743 - val_accuracy: 0.7047 - val_loss: 1.0823
Epoch 18/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 82s 47ms/step - accuracy: 0.8778 - loss: 0.3471 - val_accuracy: 0.7106 - val_loss: 1.1090
Epoch 19/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 74s 47ms/step - accuracy: 0.8850 - loss: 0.3249 - val_accuracy: 0.6890 - val_loss: 1.2019
Epoch 20/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 75s 48ms/step - accuracy: 0.8903 - loss: 0.3094 - val_accuracy: 0.7032 - val_loss: 1.2040
Epoch 21/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 74s 47ms/step - accuracy: 0.8963 - loss: 0.2847 - val_accuracy: 0.7088 - val_loss: 1.2257
Epoch 22/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 83s 48ms/step - accuracy: 0.9056 - loss: 0.2609 - val_accuracy: 0.7029 - val_loss: 1.2734
Epoch 23/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 83s 49ms/step - accuracy: 0.9081 - loss: 0.2533 - val_accuracy: 0.6959 - val_loss: 1.3271
Epoch 24/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 74s 47ms/step - accuracy: 0.9142 - loss: 0.2395 - val_accuracy: 0.7003 - val_loss: 1.3812
Epoch 25/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 83s 48ms/step - accuracy: 0.9218 - loss: 0.2210 - val_accuracy: 0.6968 - val_loss: 1.4646
Epoch 26/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 83s 48ms/step - accuracy: 0.9222 - loss: 0.2152 - val_accuracy: 0.6903 - val_loss: 1.4975
Epoch 27/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 74s 48ms/step - accuracy: 0.9288 - loss: 0.1986 - val_accuracy: 0.6963 - val_loss: 1.5846
Epoch 28/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 82s 47ms/step - accuracy: 0.9311 - loss: 0.1921 - val_accuracy: 0.6941 - val_loss: 1.6134
Epoch 29/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 83s 48ms/step - accuracy: 0.9338 - loss: 0.1823 - val_accuracy: 0.6879 - val_loss: 1.7273
```

```
Epoch 30/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 88s 52ms/step - accuracy: 0.9397 - loss: 0.1703 - val_accuracy: 0.6820 - val_loss: 1.7565
Epoch 31/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 75s 48ms/step - accuracy: 0.9390 - loss: 0.1717 - val_accuracy: 0.6920 - val_loss: 1.8205
Epoch 32/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 82s 48ms/step - accuracy: 0.9418 - loss: 0.1627 - val_accuracy: 0.6843 - val_loss: 1.9592
Epoch 33/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 82s 48ms/step - accuracy: 0.9432 - loss: 0.1612 - val_accuracy: 0.6961 - val_loss: 1.8492
Epoch 34/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 82s 48ms/step - accuracy: 0.9486 - loss: 0.1415 - val_accuracy: 0.6867 - val_loss: 1.9075
Epoch 35/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 82s 48ms/step - accuracy: 0.9478 - loss: 0.1450 - val_accuracy: 0.6815 - val_loss: 2.0039
Epoch 36/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 76s 48ms/step - accuracy: 0.9510 - loss: 0.1324 - val_accuracy: 0.6921 - val_loss: 2.0527
Epoch 37/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 74s 47ms/step - accuracy: 0.9546 - loss: 0.1283 - val_accuracy: 0.6928 - val_loss: 2.0784
Epoch 38/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 75s 48ms/step - accuracy: 0.9564 - loss: 0.1232 - val_accuracy: 0.6828 - val_loss: 2.2115
Epoch 39/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 82s 48ms/step - accuracy: 0.9543 - loss: 0.1278 - val_accuracy: 0.6807 - val_loss: 2.2134
Epoch 40/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 76s 49ms/step - accuracy: 0.9565 - loss: 0.1245 - val_accuracy: 0.6922 - val_loss: 2.1545
Epoch 41/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 80s 47ms/step - accuracy: 0.9591 - loss: 0.1175 - val_accuracy: 0.6827 - val_loss: 2.2706
Epoch 42/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 76s 49ms/step - accuracy: 0.9558 - loss: 0.1231 - val_accuracy: 0.6786 - val_loss: 2.3187
Epoch 43/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 80s 47ms/step - accuracy: 0.9585 - loss: 0.1165 - val_accuracy: 0.6880 - val_loss: 2.2655
Epoch 44/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 83s 48ms/step - accuracy: 0.9600 - loss: 0.1118 - val_accuracy: 0.6873 - val_loss: 2.2838
Epoch 45/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 83s 48ms/step - accuracy: 0.9670 - loss: 0.0927 - val_accuracy: 0.6856 - val_loss: 2.3869
Epoch 46/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 80s 47ms/step - accuracy: 0.9595 - loss: 0.1138 - val_accuracy: 0.6776 - val_loss: 2.4068
Epoch 47/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 82s 48ms/step - accuracy: 0.9603 - loss: 0.1153 - val_accuracy: 0.6856 - val_loss: 2.4640
Epoch 48/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 82s 48ms/step - accuracy: 0.9636 - loss: 0.1052 - val_accuracy: 0.6838 - val_loss: 2.4829
Epoch 49/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 89s 52ms/step - accuracy: 0.9668 - loss: 0.0949 - val_accuracy: 0.6827 - val_loss: 2.5944
Epoch 50/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 75s 48ms/step - accuracy: 0.9612 - loss: 0.1125 - val_accuracy: 0.6788 - val_loss: 2.5649
```

Evaluate the model

```
[ ] plt.plot(history.history['accuracy'], label='accuracy')
    plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.ylim([0.5, 1])
    plt.legend(loc='lower right')
```

```
<matplotlib.legend.Legend at 0x7ed5983873a0>
```




```
[ ] test_loss, test_acc = model.evaluate(test_images,  test_labels)
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 6s 18ms/step - accuracy: 0.6847 - loss: 2.5784
```

```
[ ] print('Test accuracy:', test_acc)
```

```
Test accuracy: 0.6787999868392944
```

Collab Link:

https://colab.research.google.com/drive/1v1AqE8sKLd4h7H1RS-vQUFayRdBdeNI0?usp=sharing

50 epochs kernel (5,5)

Create the convolutional base

```
[ ] model = models.Sequential()
    model.add(layers.Conv2D(32, (5, 5), activation='relu', input_shape=(32, 32, 3)))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(64, (5, 5), activation='relu'))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(64, (5, 5), activation='relu'))
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not pass
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Let's display the architecture of our model so far.

```
[ ] model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 32)	2,432
max_pooling2d (MaxPooling2D)	(None, 14, 14, 32)	0
conv2d_1 (Conv2D)	(None, 10, 10, 64)	51,264
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
conv2d_2 (Conv2D)	(None, 1, 1, 64)	102,464

Total params: 156,160 (610.00 KB)
Trainable params: 156,160 (610.00 KB)
Non-trainable params: 0 (0.00 B)

```
[ ] Start coding or generate with AI.
```

```
[ ] model.add(layers.Flatten())
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(10))
```

Here's the complete architecture of our model.

```
[ ] model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 32)	2,432
max_pooling2d (MaxPooling2D)	(None, 14, 14, 32)	0
conv2d_1 (Conv2D)	(None, 10, 10, 64)	51,264
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
conv2d_2 (Conv2D)	(None, 1, 1, 64)	102,464
flatten (Flatten)	(None, 64)	0
dense (Dense)	(None, 64)	4,160
dense_1 (Dense)	(None, 10)	650

Total params: 160,970 (628.79 KB)
Trainable params: 160,970 (628.79 KB)
Non-trainable params: 0 (0.00 B)

As you can see, our (4, 4, 64) outputs were flattened into vectors of shape (1024) before going through two Dense layers.

Compile and train the model

```
[ ] model.compile(optimizer='adam',
                  loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                  metrics=['accuracy'])

    history = model.fit(train_images, train_labels, epochs=50,
                        validation_data=(test_images, test_labels))
```

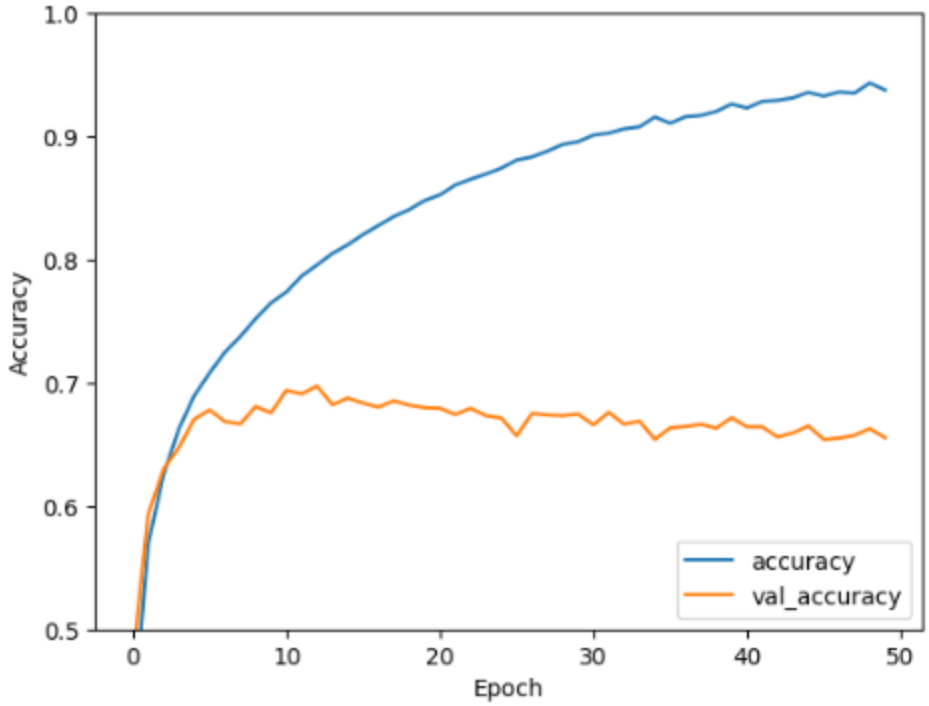
```
Epoch 1/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 14s 6ms/step - accuracy: 0.3280 - loss: 1.8093 - val_accuracy: 0.4750 - val_loss: 1.4399
Epoch 2/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 13s 3ms/step - accuracy: 0.5509 - loss: 1.2534 - val_accuracy: 0.5931 - val_loss: 1.1427
Epoch 3/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 10s 4ms/step - accuracy: 0.6166 - loss: 1.0773 - val_accuracy: 0.6299 - val_loss: 1.0554
Epoch 4/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 10s 4ms/step - accuracy: 0.6597 - loss: 0.9665 - val_accuracy: 0.6475 - val_loss: 1.0067
Epoch 5/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.6887 - loss: 0.8917 - val_accuracy: 0.6704 - val_loss: 0.9597
Epoch 6/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 9s 4ms/step - accuracy: 0.7103 - loss: 0.8243 - val_accuracy: 0.6780 - val_loss: 0.9436
Epoch 7/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.7285 - loss: 0.7718 - val_accuracy: 0.6685 - val_loss: 0.9610
Epoch 8/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 10s 4ms/step - accuracy: 0.7444 - loss: 0.7286 - val_accuracy: 0.6666 - val_loss: 0.9749
Epoch 9/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - accuracy: 0.7550 - loss: 0.6947 - val_accuracy: 0.6807 - val_loss: 0.9489
Epoch 10/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.7735 - loss: 0.6479 - val_accuracy: 0.6756 - val_loss: 0.9875
Epoch 11/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 10s 4ms/step - accuracy: 0.7783 - loss: 0.6261 - val_accuracy: 0.6939 - val_loss: 0.9557
Epoch 12/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - accuracy: 0.7955 - loss: 0.5899 - val_accuracy: 0.6910 - val_loss: 0.9589
Epoch 13/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.8027 - loss: 0.5575 - val_accuracy: 0.6973 - val_loss: 0.9676
Epoch 14/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 10s 4ms/step - accuracy: 0.8112 - loss: 0.5257 - val_accuracy: 0.6822 - val_loss: 1.0346
Epoch 15/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.8175 - loss: 0.5151 - val_accuracy: 0.6876 - val_loss: 1.0376
Epoch 16/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.8276 - loss: 0.4820 - val_accuracy: 0.6835 - val_loss: 1.0473
Epoch 17/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.8350 - loss: 0.4604 - val_accuracy: 0.6802 - val_loss: 1.0983
Epoch 18/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.8415 - loss: 0.4483 - val_accuracy: 0.6854 - val_loss: 1.1040
Epoch 19/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - accuracy: 0.8503 - loss: 0.4247 - val_accuracy: 0.6818 - val_loss: 1.1353
Epoch 20/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 11s 4ms/step - accuracy: 0.8610 - loss: 0.3933 - val_accuracy: 0.6798 - val_loss: 1.1777
Epoch 21/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 11s 4ms/step - accuracy: 0.8634 - loss: 0.3881 - val_accuracy: 0.6793 - val_loss: 1.1828
Epoch 22/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.8671 - loss: 0.3696 - val_accuracy: 0.6743 - val_loss: 1.2319
Epoch 23/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 11s 4ms/step - accuracy: 0.8739 - loss: 0.3556 - val_accuracy: 0.6793 - val_loss: 1.2807
Epoch 24/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.8798 - loss: 0.3433 - val_accuracy: 0.6733 - val_loss: 1.3591
Epoch 25/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 9s 4ms/step - accuracy: 0.8799 - loss: 0.3362 - val_accuracy: 0.6714 - val_loss: 1.3151
Epoch 26/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - accuracy: 0.8905 - loss: 0.3093 - val_accuracy: 0.6570 - val_loss: 1.4790
Epoch 27/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.8907 - loss: 0.3090 - val_accuracy: 0.6751 - val_loss: 1.4295
Epoch 28/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.8972 - loss: 0.2879 - val_accuracy: 0.6739 - val_loss: 1.4395
Epoch 29/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9008 - loss: 0.2793 - val_accuracy: 0.6733 - val_loss: 1.5146
```

```
Epoch 30/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9058 - loss: 0.2660 - val_accuracy: 0.6746 - val_loss: 1.4918
Epoch 31/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9101 - loss: 0.2538 - val_accuracy: 0.6659 - val_loss: 1.5725
Epoch 32/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9101 - loss: 0.2538 - val_accuracy: 0.6760 - val_loss: 1.6365
Epoch 33/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9151 - loss: 0.2407 - val_accuracy: 0.6664 - val_loss: 1.6703
Epoch 34/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9164 - loss: 0.2301 - val_accuracy: 0.6689 - val_loss: 1.7028
Epoch 35/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9231 - loss: 0.2180 - val_accuracy: 0.6541 - val_loss: 1.7656
Epoch 36/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 11s 4ms/step - accuracy: 0.9188 - loss: 0.2235 - val_accuracy: 0.6634 - val_loss: 1.7046
Epoch 37/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9221 - loss: 0.2198 - val_accuracy: 0.6645 - val_loss: 1.8443
Epoch 38/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - accuracy: 0.9266 - loss: 0.2055 - val_accuracy: 0.6663 - val_loss: 1.8226
Epoch 39/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9279 - loss: 0.2022 - val_accuracy: 0.6632 - val_loss: 1.8495
Epoch 40/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 10s 4ms/step - accuracy: 0.9328 - loss: 0.1896 - val_accuracy: 0.6716 - val_loss: 1.9128
Epoch 41/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9298 - loss: 0.1995 - val_accuracy: 0.6643 - val_loss: 1.9700
Epoch 42/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9345 - loss: 0.1794 - val_accuracy: 0.6642 - val_loss: 2.0152
Epoch 43/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 10s 4ms/step - accuracy: 0.9371 - loss: 0.1794 - val_accuracy: 0.6560 - val_loss: 1.9694
Epoch 44/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9394 - loss: 0.1713 - val_accuracy: 0.6593 - val_loss: 2.1247
Epoch 45/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9413 - loss: 0.1671 - val_accuracy: 0.6650 - val_loss: 2.1049
Epoch 46/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - accuracy: 0.9358 - loss: 0.1814 - val_accuracy: 0.6540 - val_loss: 2.1403
Epoch 47/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9432 - loss: 0.1579 - val_accuracy: 0.6551 - val_loss: 2.1498
Epoch 48/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9438 - loss: 0.1611 - val_accuracy: 0.6573 - val_loss: 2.1355
Epoch 49/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9522 - loss: 0.1435 - val_accuracy: 0.6628 - val_loss: 2.2020
Epoch 50/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9457 - loss: 0.1549 - val_accuracy: 0.6554 - val_loss: 2.2468
```

Evaluate the model

```
[ ] plt.plot(history.history['accuracy'], label='accuracy')
    plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.ylim([0.5, 1])
    plt.legend(loc='lower right')
```

```
<matplotlib.legend.Legend at 0x7dee58c65e10>
```




```
[ ] test_loss, test_acc = model.evaluate(test_images,  test_labels)
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.6629 - loss: 2.1980
```

```
[ ] print('Test accuracy:', test_acc)
```

```
Test accuracy: 0.6553999781608582
```

Collab Link:

https://colab.research.google.com/drive/1_lkmMYqZRYDczt4GcObCjow2AhLg8vOu?usp=sharing

Epochs and Kernel Value	Accuracy
10 (3,3)	0.71509
10 (5,5)	0.67500
20 (3,3)	0.69959
20 (5,5)	0.67540
50 (3,3)	0.67879
50 (5,5)	0.65539

- *How does the accuracy change?*

Accuracy Decreases with Higher Epochs

For **(3,3)** kernels, accuracy drops from **0.71509 (10 epochs)** to **0.67879 (50 epochs)**, For **(5,5)** kernels, accuracy decreases from **0.67500 (10 epochs)** to **0.65539 (50 epochs)**. This suggests overfitting might be occurring as the model trains for more epochs. Overfitting happens when the model learns patterns specific to the training data, making it perform poorly on unseen test data.

- *Why Accuracy is getting worse?*

With prolonged training, **the model overfits on the training data**, as evident from the decreased test accuracy. Overfitting is more pronounced with larger kernels since they increase the model's complexity.

Smaller kernels like **the (3,3) kernel consistently performs better than the (5,5) kernel** as (3,3) require fewer parameters and are better suited for learning small patterns, leading to better generalization. Larger kernels like (5,5) are more computationally expensive and prone to overfitting due to capturing broader but less focused patterns.

Week – 11 – Supervised Learning (Regression)

The **least square method** is the process of finding the best-fitting curve or line of best fit for a set of data points by reducing the sum of the squares of the offsets (residual part) of the points from the curve.
During the process of finding the relation between two variables, the trend of outcomes is estimated quantitatively. This process is termed as **regression analysis.**

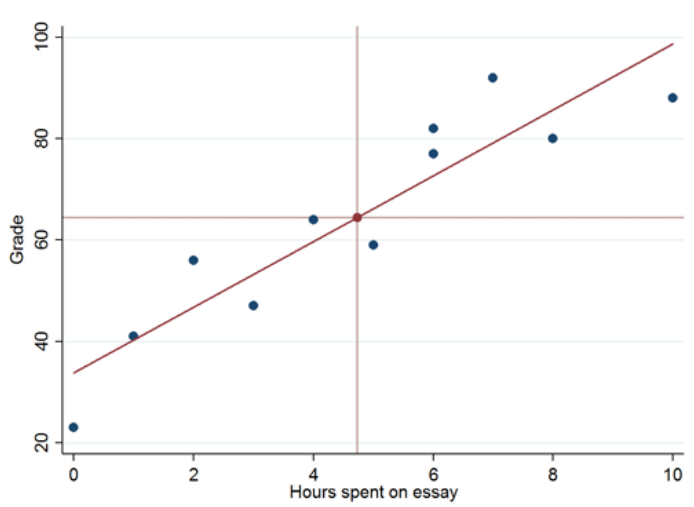
Collab Link

https://colab.research.google.com/drive/1jXpI560zNKUlmBjlMs7h457KP7SDvNrQ?usp=sharing

Task1

A teacher is interested to work out how time spent writing a logbook affects the grades. He develops a regression model to predict what score someone will get from a logbook based on how long they spent on it.
He has the data shown on the opposite:

	Hours spent on essay	Grade
	6	82
	10	88
	2	56
	4	64
	6	77
	7	92
	0	23
	1	41
	8	80
	5	59
	3	47
Mean	4.72	64.45



The best regression value is found to be:

y = 6.49x + 30.18

Calculate the Least Squares Error for this model.

To calculate the Least Squares Error, we need to first calculate predicted grades, compute the errors and the squared errors.

From the graph, we found, x (hours spent on essay) and y (actual grade), the data is provided below:

x (hours spent on essay)	y (actual grade)
6	82
10	88
2	56
4	64
6	77
7	92
0	23
1	41
8	80
5	59
3	47

- **How to calculate $y_{(prediction)}$?**

Substituting x values in $y = 6.49x + 30.18$

- **Error Calculation Formula**

Error = $y_{(actual)} - y_{(predicted)}$

- **Squared Error Formula**

Squared Error = $(\text{Error})^2$

- **Least Squares Error (LSE) Formula**

LSE = $\sum$ (Squared Errors)

x	$y_{(a)}$	$y_{(predicted)}$	Error	Squared Error
6	82	6.49 (6) + 30.18 = 69.12	82 – 69.12 = 12.88	165.89
10	88	6.49 (10) + 30.18 = 95.08	88 – 95.08 = -7.08	50.126
2	56	6.49 (2) + 30.18 = 43.16	56 – 43.16 = 12.84	164.86
4	64	6.49 (4) + 30.18 = 56.14	64 – 56.14 = 7.86	61.79
6	77	6.49 (6) + 30.18 = 69.12	77 – 69.12 = 7.88	62.09
7	92	6.49 (7) + 30.18 = 75.61	92 – 75.61 = 16.39	268.63
0	23	6.49 (0) + 30.18 = 30.18	23 – 30.18 = -7.18	51.55
1	41	6.49 (1) + 30.18 = 36.67	41 – 36.67 = 4.33	18.75
8	80	6.49 (8) + 30.18 = 82.10	80 – 82.10 = -2.10	4.41
5	59	6.49 (5) + 30.18 = 62.63	59 – 62.63 = -3.63	13.17
3	47	6.49 (3) + 30.18 = 49.65	47 – 49.65 = -2.65	7.02

Total Least Squares Error:

LSE = 165.89 + 50.126 + 164.86 + 61.79 + 62.09 + 268.63 + 51.55 + 18.75 + 4.41 + 13.18 + 7.02 = **868.29**

Task 2

Table below shows some data from the early days of the Italian clothing company Benetton. Each row in the table shows Benetton's sales for a year and the amount spent on advertising that year.
In this case, our outcome of interest is sales—it is what we want to predict. If we use advertising as the predictor variable, we can find a linear regression model which would give estimates of the sales.
Which one of the following will the best regression model? Justify your answer.

- Sales = 22.60 × Advertising + 169.2
- Sales = 23.42 × Advertising + 167.7
- Sales = 23.10 × Advertising + 168.1

Year	Sales (Million Euro)	Advertising (Million Euro)
1	651	23
2	762	26
3	856	30
4	1,063	34
5	1,190	43
6	1,298	48
7	1,421	52
8	1,440	57
9	1,518	58

Ans: We are given three regression models to compare, we also have the data for **Advertising (x)** and **Sales (y_{actual})**

```
[3] import numpy as np

    # Data from the table
    advertising = np.array([23, 26, 30, 34, 43, 48, 52, 57, 58])
    sales = np.array([651, 762, 856, 1063, 1190, 1298, 1421, 1440, 1518])

    # Models to evaluate
    models = [
        {"equation": "Sales = 22.60 × Advertising + 169.2", "coef": 22.60, "intercept": 169.2},
        {"equation": "Sales = 23.42 × Advertising + 167.7", "coef": 23.42, "intercept": 167.7},
        {"equation": "Sales = 23.10 × Advertising + 168.1", "coef": 23.10, "intercept": 168.1},
    ]

    # Function to calculate predictions, errors, and sum of squared errors
    def calculate_sse(model, advertising, sales):
        # Calculate predicted sales
        predicted_sales = model["coef"] * advertising + model["intercept"]
        # Calculate errors
        errors = sales - predicted_sales
        # Square the errors
        squared_errors = errors ** 2
        # Calculate Sum of Squared Errors
        sse = np.sum(squared_errors)
        return predicted_sales, errors, squared_errors, sse

    # Evaluate each model
    for model in models:
        predicted_sales, errors, squared_errors, sse = calculate_sse(model, advertising, sales)
        print(f"\nModel: {model['equation']}")
        print(f"Predicted Sales: {predicted_sales}")
        print(f"Errors: {errors}")
        print(f"Squared Errors: {squared_errors}")
        print(f"Sum of Squared Errors (SSE): {sse:.2f}")
```

```
Model: Sales = 22.60 × Advertising + 169.2
Predicted Sales: [ 689.   756.8  847.2  937.6 1141.  1254.  1344.4 1457.4 1480. ]
Errors: [-38.    5.2   8.8 125.4  49.   44.   76.6 -17.4  38. ]
Squared Errors: [ 1444.      27.04    77.44 15725.16  2401.    1936.    5867.56   302.76
  1444.  ]
Sum of Squared Errors (SSE): 29224.96

Model: Sales = 23.42 × Advertising + 167.7
Predicted Sales: [ 706.36  776.62  870.3   963.98 1174.76 1291.86 1385.54 1502.64 1526.06]
Errors: [-55.36 -14.62 -14.3   99.02  15.24   6.14  35.46 -62.64  -8.06]
Squared Errors: [3064.7296  213.7444  204.49   9804.9604  232.2576   37.6996 1257.4116
 3923.7696   64.9636]
Sum of Squared Errors (SSE): 18804.03

Model: Sales = 23.10 × Advertising + 168.1
Predicted Sales: [ 699.4  768.7  861.1  953.5 1161.4 1276.9 1369.3 1484.8 1507.9]
Errors: [-48.4  -6.7  -5.1 109.5  28.6  21.1  51.7 -44.8  10.1]
Squared Errors: [ 2342.56    44.89    26.01 11990.25   817.96   445.21  2672.89  2007.04
   102.01]
Sum of Squared Errors (SSE): 20448.82
```

Final Results

LSE (Model 1)	29,224.96
LSE (Model 2)	18,804.03
LSE (Model 3)	20,448.82

Based on the Final Results we can say that the regression model with the lowest LSE, **LSE (Model 2) = 18,804.03** is preferred and hence the **Best Regression Model**

Task 3

For this task, download Regression_Advertising.csv from Blackboard. This is a simple dataset that contains, in unit of £, the marketing spend along with the company sales for each month for a Software company.
Using the advertising data, model the linear relationship between the marketing budget and sales. Then answer the following questions:

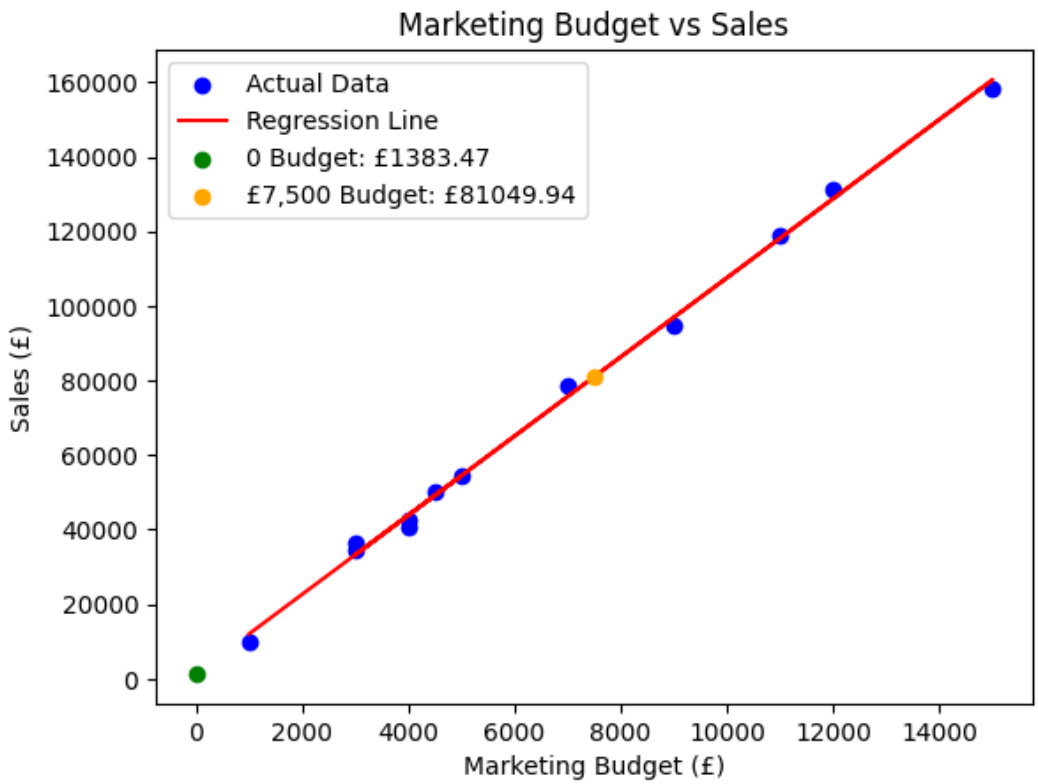
- How much sale can we expect when the marketing budget is zero?
- For a marketing spend of £7,500, how much sale can we expect?

Ans:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

# Mount Google Drive
from google.colab import drive
drive.mount('/content/gdrive')

# Define folder location
FolderName = '/content/gdrive/My Drive/Colab Notebooks/UWL/ML_L6/'

# Load the dataset
points = np.genfromtxt(FolderName + 'Regression_Advertising.csv', delimiter=',', skip_header=0)

# Extract the features (marketing spend) and target (sales)
X = np.array(points[:,0].reshape(-1, 1))
y = np.array(points[:,1].reshape(-1, 1))

# Create and train the linear regression model
model = LinearRegression()
model.fit(X, y)

print('line equation: y = mx + b')
print('optimized m:', model.coef_[0][0])
print('optimized b:', model.intercept_[0])

# 1) Zero Marketing Budget:
predicted_sales_at_zero = model.intercept_[0]
print(f"Predicted sales for zero marketing budget: £{predicted_sales_at_zero:.2f}")

# 2) Marketing Budget of £7,500:
predicted_sales_at_7500 = model.predict([[7500]])[0][0]
print(f"Predicted sales for £7,500 marketing budget: £{predicted_sales_at_7500:.2f}")

# Visualize the data and regression line
plt.scatter(X, y, color='blue', label='Actual Data')
plt.plot(X, model.predict(X), color='red', label='Regression Line')
```

```
# Add specific points for 0 and 7,500 marketing budget
plt.scatter(0, predicted_sales_at_zero, color='green', zorder=5, label=f'0 Budget: £{predicted_sales_at_zero:.2f}')
plt.scatter(7500, predicted_sales_at_7500, color='orange', zorder=5, label=f'£7,500 Budget: £{predicted_sales_at_7500:.2f}')

# Label the axes and add the title
plt.xlabel('Marketing Budget (£)')
plt.ylabel('Sales (£)')
plt.title('Marketing Budget vs Sales')

# Show legend and plot
plt.legend()
plt.show()
```

```
Drive already mounted at /content/gdrive; to attempt to forcibly remount, call drive.mount
line equation: y = mx + b
optimized m: 10.622195457815232
optimized b: 1383.4713801253529
Predicted sales for zero marketing budget: £1383.47
Predicted sales for £7,500 marketing budget: £81049.94
```



- **How much sale can we expect when the marketing budget is zero?**

We can expect sale of around 1383.47 when the Marketing Budget is £ 0.

- **For a marketing spend of £7,500, how much sale can we expect?**

We can expect sale of around 81049.94 when the Marketing Budget is £ 7,500.

Task 4

Modify the Python script to compute the cost function J for the flitted line in Task 3. Provide a snapshot of your additions to the script. Also provide the results.

Ans: The **J - cost function** is defined **as the sum of squared errors** across all training examples, and its value indicates **how well the model fits the data**.
It measures the difference between the predicted values of the model and the actual target values and hence by minimizing this cost function, we can determine the optimal values for the model's parameters and improve its performance.

```
[18] def jcost_function(x, y, m, b):
         return (1 / (2 * len(x))) * np.sum((y - (m * x + b)) ** 2)

     # Extract slope and intercept
     m, b = model.coef_[0][0], model.intercept_[0]

     # Calculate and print the cost function
     J = jcost_function(X.flatten(), y.flatten(), m, b)
     print(f"J Cost Function : {J:.2f}")
```

```
J Cost Function : 2229848.94
```