# ECE 457B Assignment 4

Tzu-Chun Chou
20718966
t6chou@uwaterloo.ca

# Q1

In [ ]:
```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder, MinMaxScaler
from sklearn.svm import SVC
from sklearn import metrics
from sklearn.metrics import confusion_matrix, classification_report, ConfusionMatrixDisplay
from sklearn.preprocessing import MinMaxScaler
```

# 1.a Data Preprocessing

In [ ]:
```
# data from white wine and red wine is already merged into a single winequality datasheet
df = pd.read_csv("winequality.csv")
print(df.head())
print(df.shape)
```

```
   fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
0            7.4              0.70         0.00             1.9      0.076
1            7.8              0.88         0.00             2.6      0.098
2            7.8              0.76         0.04             2.3      0.092
3           11.2              0.28         0.56             1.9      0.075
4            7.4              0.70         0.00             1.9      0.076

   free sulfur dioxide  total sulfur dioxide  density    pH  sulphates  \
0                 11.0                  34.0   0.9978  3.51       0.56
1                 25.0                  67.0   0.9968  3.20       0.68
2                 15.0                  54.0   0.9970  3.26       0.65
3                 17.0                  60.0   0.9980  3.16       0.58
4                 11.0                  34.0   0.9978  3.51       0.56

   alcohol  quality  type
0      9.4        5     0
1      9.8        5     0
2      9.8        5     0
3      9.8        6     0
4      9.4        5     0
(6497, 13)
```

In [ ]:
```
# column 1 -> 11 = features, column 12 = quality, column 13 = wine category
(x, y) = df.iloc[:, 0:10], df.iloc[:, 11]

# normalization
n_x = MinMaxScaler().fit_transform(x)

# 80% for training, 20% for testing
x_train, x_test, y_train, y_test = train_test_split(n_x, y, test_size=0.20)

print('Training set shape: {}'.format(x_train.shape))
print('Training labels shape: {}'.format(y_train.shape))
print('Test set shape: {}'.format(x_test.shape))
print('Test label shape: {}'.format(y_test.shape))
```

```
Training set shape: (5197, 10)
Training labels shape: (5197,)
Test set shape: (1300, 10)
Test label shape: (1300,)
```

# Support Vector Machine

In [ ]:
```
def SVM(C, kernal, g="scale"):
    svm = SVC(C=C, kernel=kernal, gamma=g)
    svm.fit(x_train, y_train)

    y_pred = svm.predict(x_test)
    acc_svm = metrics.accuracy_score(y_test, y_pred)
    print('SVM Accuracy for {} at C = {} and gamma = {}:'.format(kernal, C, g), acc_svm)

    # cm_svm = confusion_matrix(y_test, y_pred)
    # class_names = ['0', '1', '2', '3', '4', '5', '6']

    # print(cm_svm)
    # print(classification_report(y_test, y_pred, target_names=class_names))
    # disp1 = ConfusionMatrixDisplay(confusion_matrix=cm_svm)
    # disp1.plot()
    return acc_svm
```

In [ ]:
```
C1 = [1, 10, 50, 100]
C2 = [1, 10, 20, 30]

result = np.empty((3, 4), dtype=float)

# RBF Kernal
for idc, C in enumerate(C1):
    acc_svm = SVM(C, 'rbf')
    result[0, idc] = acc_svm

# Poly Kernal
for idc, C in enumerate(C1):
    acc_svm = SVM(C, 'poly')
    result[1, idc] = acc_svm

# Linear Kernal
for idc, C in enumerate(C2):
    acc_svm = SVM(C, 'linear')
    result[2, idc] = acc_svm
```

```
SVM Accuracy for rbf at C = 1 and gamma = scale: 0.546923076923077
SVM Accuracy for rbf at C = 10 and gamma = scale: 0.5930769230769231
SVM Accuracy for rbf at C = 50 and gamma = scale: 0.5838461538461538
SVM Accuracy for rbf at C = 100 and gamma = scale: 0.5892307692307692
SVM Accuracy for poly at C = 1 and gamma = scale: 0.5592307692307692
SVM Accuracy for poly at C = 10 and gamma = scale: 0.5738461538461539
SVM Accuracy for poly at C = 50 and gamma = scale: 0.573076923076923
SVM Accuracy for poly at C = 100 and gamma = scale: 0.5746153846153846
SVM Accuracy for linear at C = 1 and gamma = scale: 0.5069230769230769
SVM Accuracy for linear at C = 10 and gamma = scale: 0.5269230769230769
SVM Accuracy for linear at C = 20 and gamma = scale: 0.5238461538461539
SVM Accuracy for linear at C = 30 and gamma = scale: 0.5261538461538462
```

Accuracy result in table, the column represents the different regularization parameters used and the rows represents the different kernal used

In [ ]:
```
# create the dataframe
df = pd.DataFrame(result, columns=["C[0]", "C[1]", "C[2]", "C[3]"],
                  index=["RBF", "Poly", "Linear"])
```

```
# print the dataframe
print(df)
```

```
            C[0]      C[1]      C[2]      C[3]
RBF     0.546923  0.593077  0.583846  0.589231
Poly    0.559231  0.573846  0.573077  0.574615
Linear  0.506923  0.526923  0.523846  0.526154
```

# 1.b

Comparing the accuracy result from the table above, we can see that RBF kernal performed the best overall with an average accuracy of 57.826925% compared to 57.019225% for poly kernal and only 52.12115% for linear kernal. This make sense since guassian kernal is generally the perfered function in svm. It is suitable for non-linear data and helps to make proper separation when there is no prior knowledge of data. On the otherhand the linear kernal is the most basic kernal and is mostly preferred for text-classification and linear kernal is just a more generalized representation of the linear kernal. Furthermore, we can see that increasing the regularization parameter c sees improvement in training and test accuracy as well, since a higher value of the regularization parameter will penalize the model more for misclassifying training examples and lead to a smaller margin, while a lower value of the regularization parameter will allow more margin violations and lead to a larger margin.

# 1.c Improving the model

In [ ]:
```
C = [1, 10, 50, 100]
gamma = [10**-1, 10**0, 10*1, 10**2]

result = np.empty((len(C), len(gamma)), dtype=float)

for idc, c in enumerate(C):
    for idg, g in enumerate(gamma):
        acc_svm = SVM(c, 'rbf', g)
        result[idc, idg] = acc_svm
```

```
SVM Accuracy for rbf at C = 1 and gamma = 0.1: 0.49538461538461537
SVM Accuracy for rbf at C = 1 and gamma = 1: 0.5169230769230769
SVM Accuracy for rbf at C = 1 and gamma = 10: 0.5776923076923077
SVM Accuracy for rbf at C = 1 and gamma = 100: 0.6215384615384615
SVM Accuracy for rbf at C = 10 and gamma = 0.1: 0.5246153846153846
SVM Accuracy for rbf at C = 10 and gamma = 1: 0.5284615384615384
SVM Accuracy for rbf at C = 10 and gamma = 10: 0.5884615384615385
SVM Accuracy for rbf at C = 10 and gamma = 100: 0.6376923076923077
SVM Accuracy for rbf at C = 50 and gamma = 0.1: 0.5223076923076924
SVM Accuracy for rbf at C = 50 and gamma = 1: 0.5453846153846154
SVM Accuracy for rbf at C = 50 and gamma = 10: 0.5953846153846154
SVM Accuracy for rbf at C = 50 and gamma = 100: 0.64
SVM Accuracy for rbf at C = 100 and gamma = 0.1: 0.5269230769230769
SVM Accuracy for rbf at C = 100 and gamma = 1: 0.5592307692307692
SVM Accuracy for rbf at C = 100 and gamma = 10: 0.6046153846153847
SVM Accuracy for rbf at C = 100 and gamma = 100: 0.64
```

Accuracy result for rbf kernal in table, the column represents the different regularization parameters used and the rows represents the different gamma parameters used.

In [ ]:
```
df = pd.DataFrame(result, columns=gamma,
                  index=C)
```

```
# print the dataframe
print(df)
```

```
        0.1       1.0       10.0      100.0
1    0.495385  0.516923  0.577692  0.621538
10   0.524615  0.528462  0.588462  0.637692
50   0.522308  0.545385  0.595385  0.640000
100  0.526923  0.559231  0.604615  0.640000
```

Here we see that not only does a larger regularization parameter kernal improves the model, but also does a larger gamma parameter. In general a high gamma value may lead to overfitting, where the model captures noise in the training data and performs poorly on new, unseen data. A low gamma value may lead to underfitting, where the model is too simple to capture the underlying patterns in the data and performs poorly on both the training and testing data. Here since the number of data in the wine-dataset is very limited and consist of a low number of feature, a higher gamma is more suitable in order to prevent underfitting. Here we reached the optimal accuracy of 0.64 using C = 100 and gamma = 100

# Q.2

In [ ]:
```
from __future__ import division
from __future__ import print_function
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
import tensorflow.keras as keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, MaxPool2D, Flatten, Dropout
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical
```

## Data Preprocessing

In [ ]:
```
(train_images, train_labels), (test_images, test_labels) = cifar10.load_data()
train_labels = to_categorical(train_labels, num_classes=10)
test_labels = to_categorical(test_labels, num_classes=10)
print('train images shape:', train_images.shape)
print('train labels shape:', train_labels.shape)
print('test images shape:', test_images.shape)
print('test labels shape:', test_labels.shape)
```

```
train images shape: (50000, 32, 32, 3)
train labels shape: (50000, 10)
test images shape: (10000, 32, 32, 3)
test labels shape: (10000, 10)
```

## Data Normalization

In [ ]:
```
train_images = train_images.astype(np.float32)
mean = np.mean(train_images)
std = np.std(train_images)
train_images = (train_images - mean) / std
test_images = (test_images - mean) / std
```

In [ ]:
```
if len(train_labels.shape)==1:
    train_labels = keras.utils.to_categorical(train_labels, num_classes=10)
    test_labels = keras.utils.to_categorical(test_labels, num_classes=10)
# print(train_labels.shape, test_labels.shape)
# print(train_labels[0:3])
# print('Old labels: {}'.format(np.argmax(train_labels[:3], axis=1)))

result = np.empty((3, 4), dtype=float)
```

## Multilayer Perceptron

In [ ]:
```
# define the model type (still empty)
mlp = keras.models.Sequential()
# add a layer that just flattens the input (no weights here)
mlp.add(Flatten(input_shape=(32, 32, 3)))
# add the first hidden layer with 64 neurons, an activation function of sigmoid, and an input
mlp.add(Dense(512, activation='sigmoid', input_shape=(3072,)))
# add the second hidden layer with 64 neurons and sigmoid activation function
```

```
mlp.add(Dense(512, activation='sigmoid'))
# add the output layer with 10 units and Softmax activation function
mlp.add(Dense(10, activation='softmax'))
```

In [ ]:
```
mlp.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

# Randomly sample 20% of the training set
x_train, x_test, y_train, y_test = train_test_split(train_images, train_labels, train_size=0.
mlp_h = mlp.fit(x_train, y_train, epochs=5, batch_size=32, validation_data=(test_images, test
result[0, 0] = mlp_h.history['accuracy'][-1]
result[0, 1] = mlp_h.history['val_accuracy'][-1]

# evalute test accuracy
mlp_test = mlp.evaluate(test_images,test_labels, verbose=1)
result[0, 2] = mlp_test[0]
result[0, 3] = mlp_test[1]
```

```
Epoch 1/5
313/313 [==============================] - 7s 20ms/step - loss: 1.9011 - accuracy: 0.3210 - v
al_loss: 1.8159 - val_accuracy: 0.3667
Epoch 2/5
313/313 [==============================] - 6s 19ms/step - loss: 1.6930 - accuracy: 0.4020 - v
al_loss: 1.7204 - val_accuracy: 0.3932
Epoch 3/5
313/313 [==============================] - 6s 20ms/step - loss: 1.5865 - accuracy: 0.4410 - v
al_loss: 1.6940 - val_accuracy: 0.4004
Epoch 4/5
313/313 [==============================] - 6s 20ms/step - loss: 1.4942 - accuracy: 0.4827 - v
al_loss: 1.6626 - val_accuracy: 0.4166
Epoch 5/5
313/313 [==============================] - 6s 19ms/step - loss: 1.4213 - accuracy: 0.5012 - v
al_loss: 1.6722 - val_accuracy: 0.4098
313/313 [==============================] - 1s 3ms/step - loss: 1.6722 - accuracy: 0.4098
```

# CNN1

In [ ]:
```
cnn = Sequential()
cnn.add(Conv2D(filters=64, kernel_size=(3, 3), padding='same', activation='relu', input_shape
cnn.add(Conv2D(64, kernel_size=(3, 3), padding='same', activation='relu'))
cnn.add(Flatten())
cnn.add(Dense(512, activation='sigmoid'))
cnn.add(Dense(512, activation='sigmoid'))
cnn.add(Dense(10, activation='softmax'))
```

In [ ]:
```
cnn.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

# Randomly sample 20% of the training set
x_train, x_test, y_train, y_test = train_test_split(train_images, train_labels, train_size=0.
cnn_h = cnn.fit(x_train.reshape(-1, 32, 32, 3),
        y_train,
        batch_size=32,
        epochs=5,
        validation_data=(test_images.reshape(-1, 32, 32, 3), test_labels),
        verbose=1)
result[1, 0] = cnn_h.history['accuracy'][-1]
result[1, 1] = cnn_h.history['val_accuracy'][-1]

# evalute test accuracy
cnn_test = cnn.evaluate(test_images.reshape(-1, 32, 32, 3), test_labels)
result[1, 2] = cnn_test[0]
result[1, 3] = cnn_test[1]
```

```
Epoch 1/5
313/313 [==============================] - 98s 310ms/step - loss: 1.6751 - accuracy: 0.3894 -
val_loss: 1.4280 - val_accuracy: 0.4888
Epoch 2/5
313/313 [==============================] - 97s 309ms/step - loss: 1.1765 - accuracy: 0.5817 -
val_loss: 1.3226 - val_accuracy: 0.5263
Epoch 3/5
313/313 [==============================] - 97s 311ms/step - loss: 0.6827 - accuracy: 0.7666 -
val_loss: 1.3553 - val_accuracy: 0.5463
Epoch 4/5
313/313 [==============================] - 97s 311ms/step - loss: 0.2135 - accuracy: 0.9423 -
val_loss: 1.4616 - val_accuracy: 0.5778
Epoch 5/5
313/313 [==============================] - 93s 296ms/step - loss: 0.0382 - accuracy: 0.9948 -
val_loss: 1.5138 - val_accuracy: 0.5924
313/313 [==============================] - 7s 23ms/step - loss: 1.5138 - accuracy: 0.5924
```

# CNN2

In [ ]:
```python
cnn2 = Sequential()
cnn2.add(Conv2D(filters=64, kernel_size=(3, 3), padding='same', activation='relu'
, input_shape=(32, 32, 3)))
cnn2.add(MaxPool2D())
cnn2.add(Conv2D(64, kernel_size=(3, 3), padding='same', activation='relu'))
cnn2.add(MaxPool2D())
cnn2.add(Flatten())
cnn2.add(Dense(512, activation='sigmoid'))
cnn2.add(Dropout(0.2))
cnn2.add(Dense(512, activation='sigmoid'))
cnn2.add(Dropout(0.2))
cnn2.add(Dense(10, activation='softmax'))
```

In [ ]:
```python
cnn2.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

# Randomly sample 20% of the training set
x_train, x_test, y_train, y_test = train_test_split(train_images, train_labels, train_size=0.
cnn2_h = cnn2.fit(x_train.reshape(-1, 32, 32, 3),
        y_train,
        batch_size=32,
        epochs=5,
        validation_data=(test_images.reshape(-1, 32, 32, 3), test_labels),
        verbose=1)
result[2, 0] = cnn2_h.history['accuracy'][-1]
result[2, 1] = cnn2_h.history['val_accuracy'][-1]

# evalute test accuracy
cnn2_test = cnn2.evaluate(test_images.reshape(-1, 32, 32, 3),test_labels)
result[2, 2] = cnn2_test[0]
result[2, 3] = cnn2_test[1]
```

```
Epoch 1/5
313/313 [==============================] - 18s 54ms/step - loss: 1.7539 - accuracy: 0.3548 -
val_loss: 1.4044 - val_accuracy: 0.4844
Epoch 2/5
313/313 [==============================] - 17s 53ms/step - loss: 1.3236 - accuracy: 0.5277 -
val_loss: 1.2677 - val_accuracy: 0.5428
Epoch 3/5
313/313 [==============================] - 17s 53ms/step - loss: 1.1139 - accuracy: 0.6064 -
val_loss: 1.1537 - val_accuracy: 0.5882
Epoch 4/5
313/313 [==============================] - 17s 54ms/step - loss: 0.9357 - accuracy: 0.6653 -
val_loss: 1.1154 - val_accuracy: 0.5986
Epoch 5/5
313/313 [==============================] - 17s 53ms/step - loss: 0.7308 - accuracy: 0.7481 -
val_loss: 1.1196 - val_accuracy: 0.6126
313/313 [==============================] - 3s 9ms/step - loss: 1.1196 - accuracy: 0.6126
```

# 2.a

From the table below, we can see that in general MLP performs worse than CNN both in terms of the training accuracy and test accuracy. This makes sense because CNNs are primarily used for image recognition tasks by taking advantage of the spatial structure of an image, learning features such as edges, lines, and textures. On the other hand, MLPs are more general purposed for classification, regression, and other tasks. However, an advantage of MLP is the simple architecture of MLP compared to CNN, we can see that it trains significantly faster than CNN in general.

In [ ]:
```
df = pd.DataFrame(result, columns=["train accuracy", "train val accuray", "test loss", "test
                  index=["MLP", "CNN", "CNN2"])

print(df)
```

```
      train accuracy  train val accuray  test loss  test accuracy
MLP           0.5012             0.4098   1.672189         0.4098
CNN           0.9948             0.5924   1.513774         0.5924
CNN2          0.7481             0.6126   1.119578         0.6126
```

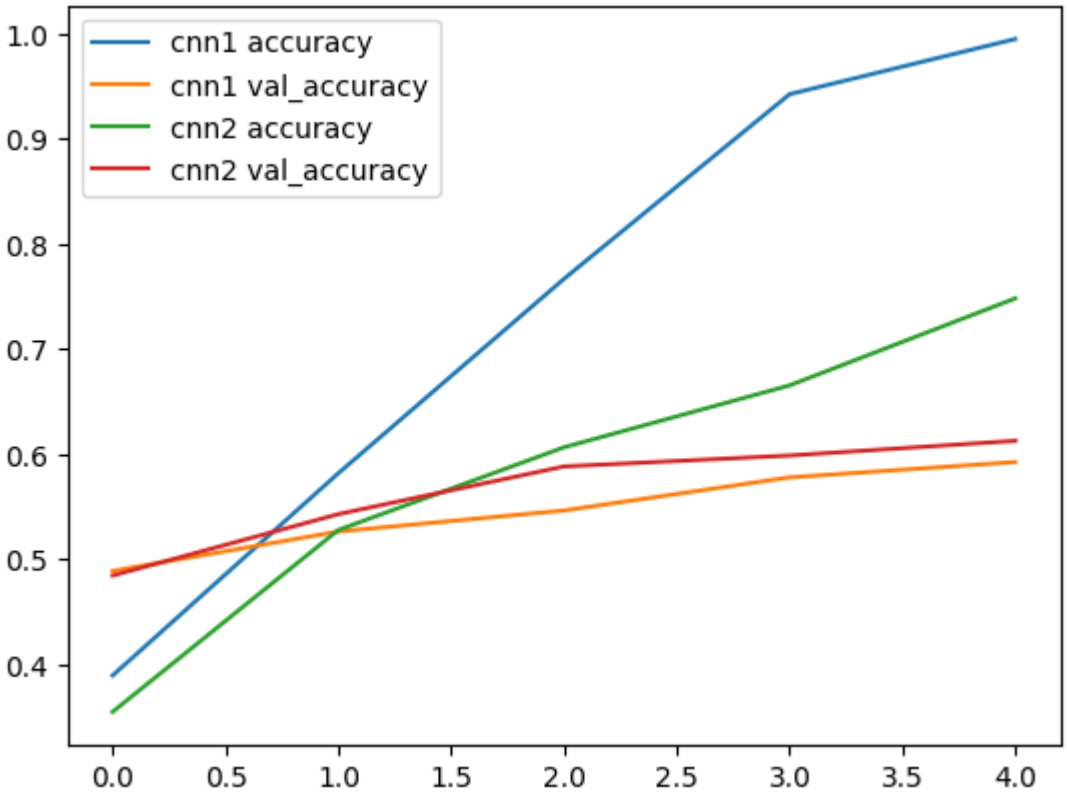
# 2.b

As we can see from the figure below, the training time for the first CNN model is significantlylonger than then the second one. This is because the 2 addtional max pooling layer reduces the size of the images by half. Between the two CNN we can see that although the first CNN shows signficantly higher training accuracy, it shows slightly lower validation accuracy and subsequently lower test accuracy, a clear sign of overfitting. This is because it lacks the dropout layer from the second CNN model which provides regularization to reduce a overfitting.

In [ ]:
```
# plot accuracy for the first CNN model
plt.plot(cnn_h.history['accuracy'], label='cnn1 accuracy')
plt.plot(cnn_h.history['val_accuracy'], label='cnn1 val_accuracy')

# plot accuracy for the second CNN model
plt.plot(cnn2_h.history['accuracy'], label='cnn2 accuracy')
plt.plot(cnn2_h.history['val_accuracy'], label='cnn2 val_accuracy')

plt.legend(loc='best')
plt.show()
```


cnn1 accuracy
cnn1 val_accuracy
cnn2 accuracy
cnn2 val_accuracy
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0