1. **Is overfitting more likely to happen when the hypothesis space is larger or smaller? List your reasons.**

Overfitting is more likely to happen when the hypothesis space is larger. The hypothesis space is the set of all possible models or functions that can be used to fit the data. When the hypothesis space is larger, the model can potentially fit the training data more closely, but it may fail to generalize well to new, unseen data.

Here are some reasons why overfitting is more likely to happen when the hypothesis space is larger:

I. A larger hypothesis space typically means that the model is more complex and has more parameters. More parameters allow the model to fit the training data more closely, but they also increase the risk of overfitting.

II. More relevant and uncorrelated features (# columns) may be added for larger hypothesis space, but # observations i.e., # rows remain same. Then the curse of dimensionality happens those results in model's poor performance.

III. With a larger hypothesis space, more data is needed to constrain the model and prevent overfitting. If the training data is insufficient, the model may fit the noise in the data and fail to generalize well.

2. **How would you identify a model with high bias?**

A model with high bias is typically one that is too simple and not able to capture the relationship of the data. There are several factors that represent high bias in data:

I. Model with high error rate on the training data.

II. If the model is underfitting i.e., is not able to capture the patterns and relationships in the data.

III. If the model's accuracy is significantly lower than other models that have been trained on the same data.

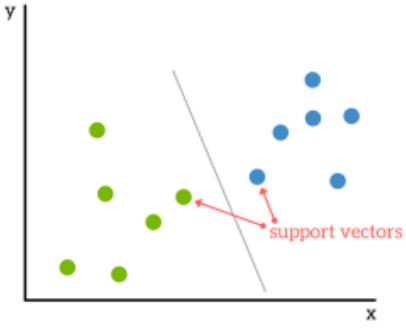
IV. If the model's performance does not improve with the increase of training data.

V. If the model has a high training error and a high validation error, indicating that it is not able to generalize well to new data.

If we observe any of the above factors, it is likely that the mode has high bias.

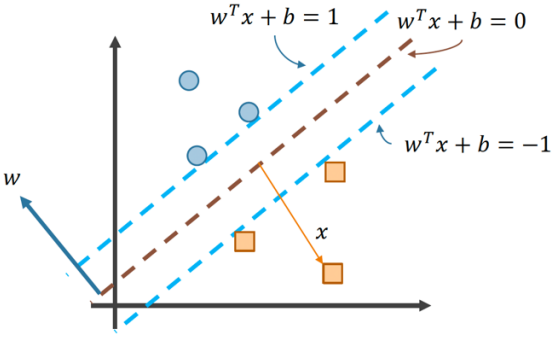
3. **What is the main idea of an SVM? Please give a short, generalized description based on your understanding with an example.**

SVM is a supervised machine learning algorithm that can be used for classification and regression tasks. It is mainly used for classification. The main concept of SVMs is to find a hyperplane that can optimally separate a dataset into two classes. The image (Ref: 1) below illustrates this concept.



Here, support vectors in SVM are the data points that are closest to the hyperplane. They are important because if they are removed from the dataset, the position of the hyperplane will be changed. Beside that the hyperplane is chosen so as to maximize the margin, which is the distance between the hyperplane and the closest data points from both classes. A hyperplane is determined by (w, b):

- $\boldsymbol{w}$ a normal vector and is orthogonal to the hyperplane, shows the hyperplane direction
- b is a scale value (bias), that determines the distance between the hyperplane and the origin point



Suppose we have a labeled dataset with two classes: positive class (+1) and negative class (-1), and the dataset has the following points:

Class (+1): (1,2), (2,3), (3,3)

Class (-1): (3,1), (4,2), (5,1)

a. First, we need to choose a kernel function to transform the data into a higher dimensional space where it is linearly separable. Let's choose a linear kernel, which means that we don't need to transform the data.
b. Next, we train the SVM on the dataset using the kernel function. The trained SVM will find the optimal hyperplane that separates the two classes.
c. We can identify the support vectors, which are the data points closest to the optimal hyperplane. In this example, the support vectors are: (1,2), (2,3), (3,3), (3,1), (5,1)
d. We can then calculate the Lagrange multipliers αi for each support vector. Suppose following
    a. values of αi for the support vectors are:
    b. $\alpha 1 = 0.342\ \alpha 2 = 0.132\ \alpha 3 = 0.474\ \alpha 4 = 0.177\ \alpha 5 = 0.0$
e. Using the formula $w = \sum \alpha i\ yi\ xi$, we can calculate the value of "w".
    a. $w = \alpha 1 * y1 * x1 + \alpha 2 * y2 * x2 + \alpha 3 * y3 * x3 + \alpha 4 * y4 * x4 + \alpha 5 * y * x5$
    b. where x1, x2, x3, x4, and x5 are the support vectors, and y1, y2, y3, y4, and y5 are their corresponding labels (+1 or -1).
    c. Substituting the values, we get:
f. $w = 0.342 * 1 * (1,2) + 0.132 * 1 * (2,3) + 0.474 * 1 * (3,3) + 0.177 * (-1) * (3,1) + 0.0 * (-1) * (5,1)$
g. $w = (1.638, 1.83)$
    a. So, the value of "w" is $(1.638, 1.83)$.
h. Finally, we know $b = yk - wT\ xk$
i. where xk is any of the support vectors, and yk is its corresponding label (+1 or -1).
j. Suppose we choose the support vector (3,1), which has a label of -1.

$$b = (-1) - (1.638, 1.83)T\ (3,1)$$

$$b = -5.382$$

So, the value of "b" is -5.382.

Therefore, the equation of the optimal hyperplane is: $wT\ x + b = 0$ which, in this case, is: $1.638\ x1 + 1.83\ x2 - 5.382 = 0$

This hyperplane separates the positive and negative classes and can be used to classify new data points.

4. **After initialization, we found our convolutional neural network's training loss does not go down, list one way to fix this problem, and briefly explain.**

To fix the problem, we can use data augmentation during training because it allows to boost the size of your dataset even further.

Data augmentation involves creating new training examples by applying various transformations to the existing data. This can help to increase the size of the training set, which can be beneficial in preventing overfitting and improving generalization. By generating new examples that are similar to the original data but with small variations, data augmentation can also help to expose the model to a wider range of inputs and improve its ability to handle variations in the data. In the case of a CNN where the training loss is not going down after initialization, data augmentation may help by providing more diverse training examples that can help the model to better capture the underlying patterns in the data. This can help to prevent the model from getting stuck in a suboptimal solution and help it to converge to a better solution.

5. **Train a polynomial SVM or an RBF Kernel for the Iris dataset. Use a train-test 80% to 20% balanced split (include the train and test sets you created with your submission), specify any parameter settings used, include your choice and rationale for it. Discuss the performance of the model you trained.**

```
# %%
import pandas as pd
from sklearn.model_selection import train_test_split

# Load the Iris dataset
iris = pd.read_csv('Iris.csv')
X = iris.drop(['Species'], axis = 1)
y = iris.Species

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y,
random_state=42)

# train set
train_set = pd.concat([X_train, y_train], axis=1)
train_set.to_csv('train_iris')
# test set
test_set = pd.concat([X_test, y_test], axis=1)
test_set.to_csv('test_iris')

# %%
```

```
26. from sklearn.svm import SVC
27.
28. # Polynomial SVM
29. poly_svc = SVC(kernel='poly', degree=3, C=1.0)
30. poly_svc.fit(X_train, y_train)
31.
32. # %%
33. from sklearn.metrics import accuracy_score
34.
35. # Predict using the trained models
36. poly_y_pred = poly_svc.predict(X_test)
37.
38. # Calculate the accuracy of the models
39. poly_accuracy = accuracy_score(y_test, poly_y_pred)
40.
41. print(f"Polynomial SVM accuracy: {poly_accuracy:.2f}")
```

➔ The degree parameter in polynomial SVM controls the degree of the polynomial kernel function used for mapping the data to a higher-dimensional feature space. By default, the degree is set to 3 in scikit-learn's SVC implementation. We have used this default value for the degree parameter as it is a reasonable starting point and is known to work well for many datasets.

➔ The regularization parameter C in SVM controls the tradeoff between achieving a low training error and a low testing error. A small value of C creates a wider margin hyperplane but allows some misclassifications in the training set, while a large value of C creates a narrow margin hyperplane that attempts to correctly classify all training examples. We have chosen C to be 1.0, which is the default value in scikit-learn's SVC implementation. This value generally works well for many datasets and is a reasonable starting point.

➔ In these settings, my model achieved accuracy of 97% means that the polynomial SVM has correctly predicted the class labels of 97% of the test examples in the Iris dataset. In other words, out of the total number of test examples, only 3% were misclassified by the model.

```
print(f"Polynomial SVM accuracy: {poly_accuracy:.2f}")
```

✓ 0.1s

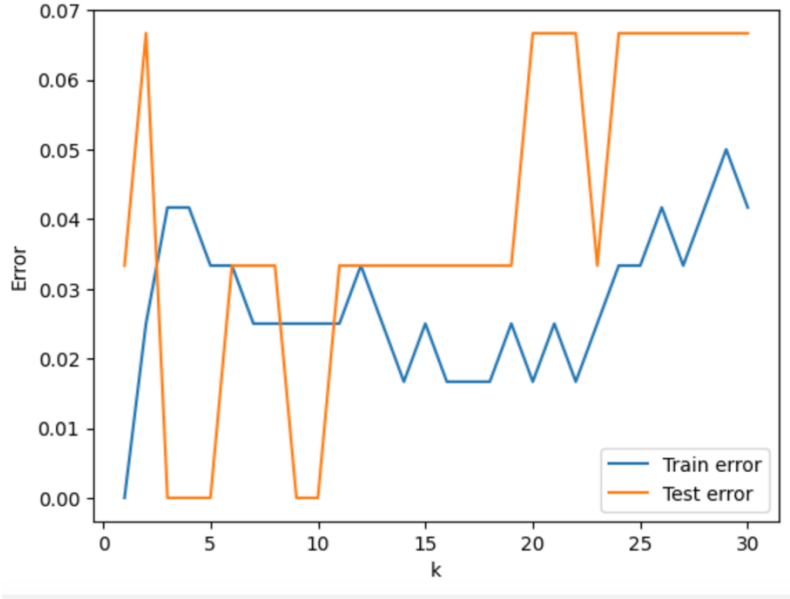
```
Polynomial SVM accuracy: 0.97
```

6.1. Use the 150 observations in the Iris dataset to train a kNN model. Use a train-test 80% to 20% balanced split, plot the train error and test error for k=1-30, what is the best k? Explain your choice. (Include the train and test sets you created with the submission.)

```
# %%
import pandas as pd
from sklearn.model_selection import train_test_split

# Load the Iris dataset
iris = pd.read_csv('Iris.csv')
iris.drop(['Id'],axis=1, inplace=True)
X = iris.drop(['Species'], axis = 1)
y = iris.Species

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=42)

# train set
train_set = pd.concat([X_train, y_train], axis=1)
train_set.to_csv('train_iris_knn')
# test set
test_set = pd.concat([X_test, y_test], axis=1)
test_set.to_csv('test_iris_knn')

# %%
from sklearn.neighbors import KNeighborsClassifier
import matplotlib.pyplot as plt

train_errors = []
test_errors = []

for k in range(1, 31):
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)
    train_errors.append(1 - knn.score(X_train, y_train))
    test_errors.append(1 - knn.score(X_test, y_test))

plt.plot(range(1, 31), train_errors, label='Train error')
plt.plot(range(1, 31), test_errors, label='Test error')
plt.xlabel('k')
plt.ylabel('Error')
plt.legend()
plt.show()

# %%
new_data = [[6.2, 3.0, 4.5, 1.8], [5.2, 3.0, 2.5, 0.9]]
```

```
knn_1 = KNeighborsClassifier(n_neighbors=1)
knn_1.fit(X_train, y_train)
print('k=1 predictions:', knn_1.predict(new_data))

knn_6 = KNeighborsClassifier(n_neighbors=6)
knn_6.fit(X_train, y_train)
print('k=6 predictions:', knn_6.predict(new_data))
```

- ➔ Here, we define a range of k values to test, ranging from 1 to 30. We then loop over each value of k, train a kNN model using the **KNeighborsClassifier()** class from scikit-learn, and evaluate its performance on both the training and test sets using the **score()** method. We calculate the error rate as 1 - accuracy and store the results in the **train_errors** and **test_errors** lists.
- ➔ To determine the best value of k, let's observe the the plot of test and train errors. Where the test error is the lowest, we considered that value of k as best. This point indicates the k value that provides the best balance between model complexity and accuracy. In our example best k value is 6.



6.2 Based on the model you created, predict the species of iris for the measurements of (6.2, 3.0, 4.5, 1.8) and (5.2, 3.0, 2.5, 0.9) which are the Sepal.Length, Sepal.Width, Petal.Length, and Petal.Width respectively, with k=1 and the best k you determined in the previous step.

```
new_data = [[6.2, 3.0, 4.5, 1.8], [5.2, 3.0, 2.5, 0.9]]
```

```
knn_1 = KNeighborsClassifier(n_neighbors=1)
knn_1.fit(X_train, y_train)
print('k=1 predictions:', knn_1.predict(new_data))

knn_6 = KNeighborsClassifier(n_neighbors=6)
knn_6.fit(X_train, y_train)
print('k=6 predictions:', knn_6.predict(new_data))
```

Output:

```
k=1 predictions: ['Iris-virginica' 'Iris-versicolor']
k=6 predictions: ['Iris-versicolor' 'Iris-setosa']
/Users/sharminsultana/opt/anaconda3/envs/machine_learnin
  warnings.warn(
/Users/sharminsultana/opt/anaconda3/envs/machine_learnin
  warnings.warn(
```

7. Using the chihuahua and muffin dataset train a simple CNN model. Split the dataset into training and testing. The train to test ratio is 8.2. Then implement a CNN model. Provide code, your train and test sets and the classification you got on your testing images.

```
# %%
import os
import numpy as np
from keras.preprocessing.image import ImageDataGenerator

# set path to the dataset folder in Google Drive
data_path = 'chihuahua-muffin'

# create data generators for training and testing sets
train_datagen = ImageDataGenerator(rescale=1./255, shear_range=0.2, zoom_range=0.2,
horizontal_flip=True, validation_split=0.2)
test_datagen = ImageDataGenerator(rescale=1./255)

# set batch size
batch_size = 32

# generate training set
train_set = train_datagen.flow_from_directory(data_path+'/train', target_size=(64, 64),
batch_size=batch_size, class_mode='binary', subset='training')

# generate validation set
```

```
val_set = train_datagen.flow_from_directory(data_path, target_size=(64, 64), batch_size=batch_size,
class_mode='binary', subset='validation')

# generate testing set
test_set = test_datagen.flow_from_directory(data_path+'/test', target_size=(64, 64),
batch_size=batch_size, class_mode='binary', shuffle=False)

# %%
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

# define CNN architecture
model = Sequential()
model.add(Conv2D(32, (3, 3), input_shape=(64, 64, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(32, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

# compile model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# train model
history = model.fit(train_set, epochs=10, validation_data=val_set)

# %%
# evaluate model on testing set
score = model.evaluate(test_set)
print('Test loss:', score[0])
print('Test accuracy:', score[1])

# get predictions for testing set
predictions = model.predict(test_set)
predicted_classes = np.round(predictions)

# print classification results
print('Classification results:')
for i, image in enumerate(test_set.filenames):
    print('{} - {}'.format(image, 'chihuahua' if predicted_classes[i] == 0 else 'muffin'))
```

Classification Result:

```
1/1 [==============================] - 0s 77ms/step - loss: 0.6850 - accuracy: 0.5000
Test loss: 0.6849626898765564
Test accuracy: 0.5
1/1 [==============================] - 0s 367ms/step
Classification results:
chihuahua/chihuahua-4.jpg - muffin
chihuahua/chihuahua-8.jpg - muffin
muffin/muffin-6.jpeg - muffin
muffin/muffin-8.jpeg - muffin
```