

SESSION 5

This session can be best explained alongwith `Session 9`

TOC:

Section I) TensorFlow

1. Introduction to TensorFlow
2. Deep Learning
3. Artificial Neural Networks(ANN)
4. Convolutional Neural Networks (CNN)
5. Recurrent Neural Networks (RNN)
6. Generative Adversarial Networks (GAN)
7. Object Detection and Tracking
8. Image Classification

Section II) Keras

1. Neural network models
2. Sequential Models
3. Functional Models
4. Model Optimization and Tuning
5. Regularization Techniques

Section III) Scikit learn

1. Regression models
2. Classification Problems
3. Decision Trees
4. Random Forest
5. Ensemble Learning
6. Scikit Learn - Project on Label Propagation

SECTION - I) TensorFlow

1. Introduction to TensorFlow

TensorFlow is an open-source machine learning framework developed by Google. It provides a comprehensive set of tools and libraries for building and deploying machine learning models. TensorFlow is known for its flexibility, scalability, and support for deep learning algorithms.

Here's an example of Python code that demonstrates the basics of TensorFlow:

```
In [ ]: import tensorflow as tf
```

This is a very simple example, but it demonstrates the basic concepts of TensorFlow. In more complex applications, you would use TensorFlow to create and train machine learning models.

```
In [ ]: import tensorflow as tf

# Create a constant tensor
x = tf.constant(5)

# Create a variable tensor
y = tf.Variable(10)

# Add the two tensors
z = tf.add(x, y)

# Create a session to run the computation
sess = tf.Session()

# Print the result
print(sess.run(z))
```

This code will print the output 15.

Here is a more detailed explanation of what the code is doing:

The first line imports the TensorFlow library. The second line creates a constant tensor with the value 5. The third line creates a variable tensor with the value 10. The fourth line adds the two tensors and stores the result in a new tensor called z. The fifth line creates a session to run the computation.

The sixth line prints the result of the computation.

2. Deep Learning

Deep learning is a subfield of machine learning that focuses on building and training neural networks with multiple layers. Deep learning models are capable of automatically learning hierarchical representations of data, leading to improved performance in tasks such as image recognition, natural language processing, and speech recognition.

Here's an example of Python code that demonstrates the construction of a deep learning model using TensorFlow:

```
In [ ]: import tensorflow as tf
        from tensorflow.keras.models import Sequential
        from tensorflow.keras.layers import Dense
```

```
In [ ]: import tensorflow as tf

        # Import the MNIST dataset
        mnist = tf.keras.datasets.mnist

        # Load the training data
        (x_train, y_train), (x_test, y_test) = mnist.load_data()

        # Normalize the data
        x_train = x_train / 255.0
        x_test = x_test / 255.0

        # Define the model
        model = tf.keras.models.Sequential([
            tf.keras.layers.Flatten(input_shape=(28, 28)),
            tf.keras.layers.Dense(128, activation='relu'),
            tf.keras.layers.Dense(10, activation='softmax')
        ])

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

        # Train the model
        model.fit(x_train, y_train, epochs=10)
```

```
# Evaluate the model
model.evaluate(x_test, y_test)
```

```
Epoch 1/10
1875/1875 [=====] - 14s 5ms/step - loss: 0.2618 - accuracy: 0.9253
Epoch 2/10
1875/1875 [=====] - 10s 5ms/step - loss: 0.1129 - accuracy: 0.9665
Epoch 3/10
1875/1875 [=====] - 10s 6ms/step - loss: 0.0774 - accuracy: 0.9772
Epoch 4/10
1875/1875 [=====] - 11s 6ms/step - loss: 0.0583 - accuracy: 0.9821
Epoch 5/10
1875/1875 [=====] - 9s 5ms/step - loss: 0.0459 - accuracy: 0.9862
Epoch 6/10
1875/1875 [=====] - 10s 5ms/step - loss: 0.0362 - accuracy: 0.9886
Epoch 7/10
1875/1875 [=====] - 10s 5ms/step - loss: 0.0282 - accuracy: 0.9916
Epoch 8/10
1875/1875 [=====] - 7s 4ms/step - loss: 0.0229 - accuracy: 0.9931
Epoch 9/10
1875/1875 [=====] - 9s 5ms/step - loss: 0.0198 - accuracy: 0.9939
Epoch 10/10
1875/1875 [=====] - 8s 4ms/step - loss: 0.0157 - accuracy: 0.9951
313/313 [=====] - 2s 2ms/step - loss: 0.0762 - accuracy: 0.9794
```

```
Out[ ]: [0.0761634036898613, 0.9793999791145325]
```

This code will create a deep learning model with two hidden layers, each with 128 neurons. The model will be trained on the MNIST dataset, which consists of 60,000 training images and 10,000 test images. The model will be trained for 10 epochs, and the accuracy will be evaluated on the test set after each epoch.

The output of the code will be the accuracy of the model on the test set. In this example, the accuracy is likely to be around 98%.

3. Artificial Neural Networks(ANN)

Artificial Neural Networks contain artificial neurons which are called units. These units are arranged in a series of layers that together constitute the whole Artificial Neural Network in a system. A layer can have only a dozen units or millions of units as this depends on how the complex neural networks will be required to learn the hidden patterns in the dataset.

Types of ANN :

4. Convolutional Neural Networks (CNN)

Convolutional Neural Networks (CNNs) are a type of deep learning model specifically designed for processing grid-like data, such as images. CNNs use convolutional layers to automatically learn spatial hierarchies of features from the input data. They have achieved remarkable success in computer vision tasks like image classification, object detection, and image segmentation.

Here's an example of Python code that demonstrates the construction and training of a Convolutional Neural Network (CNN) using TensorFlow:

```
In [ ]: import tensorflow as tf
        from tensorflow.keras.models import Sequential
        from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
```

This code can be used to train a CNN to classify images from the CIFAR10 dataset. The model can be improved by using a larger number of epochs, a different optimizer, or a different loss function. The model can also be used to classify images from other datasets

```
In [ ]: import tensorflow as tf

        # Import the CIFAR10 dataset
        (x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()

        # Normalize the data
        x_train = x_train / 255.0
        x_test = x_test / 255.0

        # Define the model
        model = tf.keras.models.Sequential([
            tf.keras.layers.Conv2D(32, (3, 3), padding='same', activation='relu', input_shape=(32, 32, 3)),
            tf.keras.layers.MaxPooling2D((2, 2), strides=(2, 2)),
            tf.keras.layers.Conv2D(64, (3, 3), padding='same', activation='relu'),
            tf.keras.layers.MaxPooling2D((2, 2), strides=(2, 2)),
            tf.keras.layers.Flatten(),
            tf.keras.layers.Dense(128, activation='relu'),
            tf.keras.layers.Dense(10, activation='softmax')
        ])

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

        # Train the model
        model.fit(x_train, y_train, epochs=10)
```

```
# Evaluate the model  
model.evaluate(x_test, y_test)
```

This code will first import the CIFAR10 dataset, which is a collection of 60,000 32x32 color images of 10 different classes. The code will then normalize the data by dividing each pixel value by 255. This is important because it ensures that all of the values in the data are between 0 and 1.

Next, the code will define the model. The model consists of 5 layers:

A convolutional layer with 32 filters of size 3x3. A max pooling layer with a pool size of 2x2. Another convolutional layer with 64 filters of size 3x3. Another max pooling layer with a pool size of 2x2. A flattening layer that converts the 2D output of the previous layer to a 1D vector. A dense layer with 128 neurons. A final dense layer with 10 neurons, one for each class. The code will then compile the model using the Adam optimizer and the categorical crossentropy loss function. The model will then be trained for 10 epochs.

Finally, the code will evaluate the model on the test set. The accuracy of the model will be printed to the console.

5. Recurrent Neural Networks (RNN)

Recurrent Neural Networks (RNNs) are a type of neural network that is well-suited for processing sequential data, such as time series or text. RNNs have recurrent connections that allow information to persist and be shared across different time steps. This enables them to capture temporal dependencies and perform tasks like speech recognition, language modeling, and sentiment analysis.

Here's an example of Python code that demonstrates the construction and training of a Recurrent Neural Network (RNN) using TensorFlow:

```
In [ ]: import tensorflow as tf  
        from tensorflow.keras.models import Sequential  
        from tensorflow.keras.layers import Embedding, LSTM, Dense
```

This code can be used to train an RNN to classify images from the MNIST dataset. The model can be improved by using a larger number of epochs, a different optimizer, or a different loss function. The model can also be used to classify images from other datasets.

```
In [ ]: import tensorflow as tf  
  
        # Import the MNIST dataset  
        (x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()  
  
        # Normalize the data
```

```
x_train = x_train / 255.0
x_test = x_test / 255.0

# Define the model
model = tf.keras.models.Sequential([
    tf.keras.layers.Embedding(784, 128),
    tf.keras.layers.LSTM(128),
    tf.keras.layers.Dense(10, activation='softmax')
])

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

# Train the model
model.fit(x_train, y_train, epochs=10)

# Evaluate the model
model.evaluate(x_test, y_test)
```

This code will first import the MNIST dataset, which is a collection of 60,000 28x28 grayscale images of handwritten digits. The code will then normalize the data by dividing each pixel value by 255. This is important because it ensures that all of the values in the data are between 0 and 1.

Next, the code will define the model. The model consists of 3 layers:

An embedding layer that converts each 28x28 image to a 128-dimensional vector. An LSTM layer with 128 units. A dense layer with 10 neurons, one for each class. The code will then compile the model using the Adam optimizer and the sparse categorical crossentropy loss function. The model will then be trained for 10 epochs.

Finally, the code will evaluate the model on the test set. The accuracy of the model will be printed to the console.

6. Generative Adversarial Networks (GAN)

Generative Adversarial Networks (GANs) are a class of deep learning models that involve two neural networks: a generator and a discriminator. The generator network generates synthetic data samples, while the discriminator network tries to distinguish between real and generated data. GANs are widely used for tasks such as image generation, image-to-image translation, and data synthesis.

Here's an example of Python code that demonstrates the implementation of a basic Generative Adversarial Network (GAN) using TensorFlow:

```
In [ ]: import tensorflow as tf
        from tensorflow.keras import layers
        import numpy as np
        import matplotlib.pyplot as plt
```

Define the generator network

```
In [ ]: def build_generator(latent_dim):
        model = tf.keras.Sequential()
        model.add(layers.Dense(256, input_dim=latent_dim, activation='relu'))
        model.add(layers.BatchNormalization())
        model.add(layers.Dense(512, activation='relu'))
        model.add(layers.BatchNormalization())
        model.add(layers.Dense(1024, activation='relu'))
        model.add(layers.BatchNormalization())
        model.add(layers.Dense(784, activation='tanh'))
        model.add(layers.Reshape((28, 28, 1)))
        return model
```

Define the discriminator network

```
In [ ]: def build_discriminator():
        model = tf.keras.Sequential()
        model.add(layers.Flatten(input_shape=(28, 28, 1)))
        model.add(layers.Dense(512, activation='relu'))
        model.add(layers.Dense(256, activation='relu'))
        model.add(layers.Dense(1, activation='sigmoid'))
        return model
```

Define the loss functions

```
In [ ]: cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)
```

Define the generator and discriminator models

```
In [ ]: latent_dim = 100
        generator = build_generator(latent_dim)
        discriminator = build_discriminator()
```


Define the optimizer for both models

```
In [ ]: generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)
```

Define the training loop

```
In [ ]: num_epochs = 50
batch_size = 128
random_dim = 100
num_examples_to_generate = 16
```

Create random noise samples for testing the generator

```
In [ ]: test_seed = tf.random.normal([num_examples_to_generate, random_dim])
```

Define a function to generate images using the generator

```
In [ ]: def generate_images(model, test_input):
    predictions = model(test_input, training=False)
    fig = plt.figure(figsize=(4, 4))
    for i in range(predictions.shape[0]):
        plt.subplot(4, 4, i+1)
        plt.imshow(predictions[i, :, :, 0] * 0.5 + 0.5, cmap='gray')
        plt.axis('off')
    plt.show()
```

Define the training loop

```
In [ ]: @tf.function
def train_step(images):
    noise = tf.random.normal([batch_size, random_dim])
    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)
        real_output = discriminator(images, training=True)
        fake_output = discriminator(generated_images, training=True)
        gen_loss = cross_entropy(tf.ones_like(fake_output), fake_output)
```

```

disc_loss_real = cross_entropy(tf.ones_like(real_output), real_output)
disc_loss_fake = cross_entropy(tf.zeros_like(fake_output), fake_output)
disc_loss = disc_loss_real + disc_loss_fake
gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.trainable_variables)
generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_variables))
discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, discriminator.trainable_variables))

```

Start the training

```

In [ ]: def train(dataset, epochs):
        for epoch in range(epochs):
            for image_batch in dataset:
                train_step(image_batch)

        # Generate images after each epoch
        generate_images(generator, test_seed)

```

Load and preprocess the MNIST dataset

```

In [ ]: (train_images, train_labels), (_, _) = tf.keras.datasets.mnist.load_data()
train_images = train_images.reshape(train_images.shape[0], 28, 28, 1).astype('float32')
train_images = (train_images - 127.5) / 127.5 # Normalize the images to [-1, 1]
train_dataset = tf.data.Dataset.from_tensor_slices(train_images).shuffle(60000).batch(batch_size)

```

Downloading data from <https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz>
11490434/11490434 [=====] - 23s 2us/step

Start the training

```

In [ ]: train(train_dataset, num_epochs)

```

In this code, we define a basic GAN for generating handwritten digit images using the MNIST dataset. The generator network takes random noise as input and generates synthetic images. The discriminator network tries to distinguish between real images from the dataset and fake images generated by the generator. The models are trained using the Adam optimizer and the binary cross-entropy loss.

The training loop consists of multiple epochs, where in each epoch, we iterate over the dataset and perform a forward and backward pass to update the generator and discriminator models. After each epoch, we generate a sample of images using the generator to visualize the progress of the GAN.

7. Object Detection and Tracking

Object detection and tracking involve identifying and locating objects within images or videos and tracking their movements across frames.

TensorFlow provides pre-trained models like the Single Shot MultiBox Detector (SSD) and the Faster R-CNN for object detection. These models can be fine-tuned on custom datasets or used directly for tasks such as object detection, instance segmentation, and object tracking.

Here's an example of Python code that demonstrates the use of TensorFlow for object detection and tracking using a pre-trained model:

```
In [ ]: import cv2
import numpy as np
import tensorflow as tf
from object_detection.utils import label_map_util
from object_detection.utils import visualization_utils as vis_util
```

This is a very simple example, but it demonstrates the basic concepts of object detection and tracking using TensorFlow. In more complex applications, you would use TensorFlow to detect and track objects in real-time, such as in self-driving cars or video surveillance systems.

```
In [ ]: import tensorflow as tf
from object_detection.utils import visualization_utils as vis_utils

# Load the pre-trained model
model = tf.saved_model.load('path/to/model')

# Create a video capture object
cap = cv2.VideoCapture('path/to/video')

# Loop over the frames in the video
while cap.isOpened():

    # Capture the current frame
    ret, frame = cap.read()

    # Convert the frame to a tensor
    image = tf.convert_to_tensor(frame)

    # Run the model on the image
    detections = model(image)

    # Visualize the detections
    vis_utils.visualize_boxes_and_labels_on_image_array(
```

```
frame,
detections['detection_boxes'],
detections['detection_classes'],
detections['detection_scores'],
category_index=None,
use_normalized_coordinates=True,
line_thickness=2,
min_score_thresh=0.5)

# Display the frame
cv2.imshow('Object Detection', frame)

# Wait for a key press
key = cv2.waitKey(1)

# If the user presses `q`, stop the loop
if key == ord('q'):
    break

# Close the video capture object
cap.release()

# Destroy all windows
cv2.destroyAllWindows()
```

This code will load a pre-trained object detection model and use it to detect objects in a video. The detected objects will be visualized on the video frame. The user can press the q key to stop the loop.

Here is a more detailed explanation of what the code is doing:

The first line imports the TensorFlow library. The second line imports the OpenCV library. The third line loads the pre-trained model. The fourth line creates a video capture object. The fifth line loops over the frames in the video. The sixth line captures the current frame. The seventh line converts the frame to a tensor. The eighth line runs the model on the image. The ninth line visualizes the detections. The tenth line displays the frame. The eleventh line waits for a key press. The twelfth line closes the video capture object. The thirteenth line destroys all windows.

8. Image Classification

Image classification is the task of assigning labels or categories to images based on their content. TensorFlow offers pre-trained models like the Inception-v3, ResNet, and MobileNet that have achieved state-of-the-art performance on popular image classification benchmarks. These models can

be used as-is or fine-tuned on custom datasets for specific image classification tasks.

Here's an example of Python code that demonstrates image classification using a pre-trained model in TensorFlow:

```
In [ ]: import tensorflow as tf
        from tensorflow.keras.applications import InceptionV3
        from tensorflow.keras.preprocessing import image
        from tensorflow.keras.applications.inception_v3 import preprocess_input, decode_predictions
        import numpy as np
```

Load the pre-trained InceptionV3 model

```
In [ ]: model = InceptionV3(weights='imagenet')
```

Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/inception_v3/inception_v3_weights_tf_dim_ordering_tf_kernels.h5
96112376/96112376 [=====] - 180s 2us/step

Load and preprocess the image

```
In [ ]: img_path = 'path/to/your/image.jpg'
        img = image.load_img(img_path, target_size=(299, 299))
        x = image.img_to_array(img)
        x = np.expand_dims(x, axis=0)
        x = preprocess_input(x)
```

Perform image classification

```
In [ ]: preds = model.predict(x)
        decoded_preds = decode_predictions(preds, top=3)[0]
```

Print the top predictions

```
In [ ]: print('Predicted:', decoded_preds)
```

Output:

Predicted: [('n02124075', 'Egyptian_cat', 0.6964707), ('n02123045', 'tabby', 0.121809535), ('n02123159', 'tiger_cat', 0.070878185)]

In this code, we first load the pre-trained InceptionV3 model using the InceptionV3 class from tensorflow.keras.applications. The model is pre-trained on the ImageNet dataset, which contains millions of labeled images across thousands of categories.

We then load and preprocess the image that we want to classify. The image is loaded using image.load_img, resized to the input size expected by the InceptionV3 model (299x299 pixels), and converted to a NumPy array. We also preprocess the input image using preprocess_input function to ensure it is formatted appropriately for the InceptionV3 model.

Next, we pass the preprocessed image through the model using model.predict. The model returns a prediction in the form of a probability distribution over the ImageNet categories. To interpret the predictions, we use decode_predictions function from tensorflow.keras.applications.inception_v3 to obtain the top predicted classes along with their corresponding labels and probabilities.

Finally, we print the top predictions. In the example output shown in the code comments, the model predicts that the image contains an Egyptian cat with a probability of 69.65%, followed by the labels "tabby" and "tiger_cat" with probabilities of 12.18% and 7.09% respectively.

SECTION - II) Keras

Keras is a high-level neural networks library written in Python that runs on top of TensorFlow, CNTK, or Theano. It provides a user-friendly and efficient interface for building and training neural network models. Keras allows for rapid prototyping and supports a wide range of neural network architectures and applications.

1. Neural Network Models:

Neural network models in Keras are built by stacking layers on top of each other. Each layer performs specific operations and transforms the input data. Keras provides a variety of layer types, including dense (fully connected) layers, convolutional layers, recurrent layers, and more. These layers can be configured and connected to form a neural network model.

2. Sequential Models:

Sequential models in Keras are a linear stack of layers, where each layer has exactly one input tensor and one output tensor. Sequential models are appropriate for building simple feedforward neural networks. Here's an example of creating a sequential model in Keras:

```
In [ ]: from tensorflow.keras.models import Sequential
        from tensorflow.keras.layers import Dense
```

3. Functional Models:

Functional models in Keras allow for more complex network architectures by allowing layers to be connected in a more flexible manner. It enables the creation of models with multiple input or output tensors, shared layers, and branching structures. Here's an example of creating a functional model in Keras:

```
In [ ]: from tensorflow.keras.models import Model
        from tensorflow.keras.layers import Input, Dense
```

4. Model Optimization and Tuning:

Keras provides a range of optimization algorithms (optimizers) that can be used to train neural network models. These optimizers adjust the model's weights based on the gradients of the loss function. Common optimizers in Keras include SGD (Stochastic Gradient Descent), Adam, RMSprop, and more. Additionally, Keras allows for customizing learning rates, momentum, and other parameters of the optimizers.

5. Regularization Techniques:

Regularization techniques in Keras help prevent overfitting, which occurs when a model performs well on the training data but fails to generalize to unseen data. Keras supports various regularization techniques, including L1 and L2 regularization, dropout, and early stopping. These techniques can be applied to individual layers or the entire model to improve generalization performance.

SECTION - III) Scikit-learn

Scikit-learn is a popular machine learning library in Python that provides a wide range of tools and algorithms for various machine learning tasks. It is built on top of NumPy, SciPy, and matplotlib and offers a user-friendly interface for training and evaluating machine learning models.

1. Regression Models:

Scikit-learn provides several regression models for predicting continuous target variables. Linear regression, polynomial regression, and support vector regression are some of the commonly used models. Here's an example of training a linear regression model in scikit-learn:

```
In [ ]: from sklearn.linear_model import LinearRegression
```

```
In [ ]: import numpy as np
import pandas as pd
from sklearn.linear_model import LinearRegression

# Load the data
df = pd.read_csv('data/stock_prices.csv')

# Split the data into features and target
X = df.drop('target', axis=1)
y = df['target']

# Create the model
model = LinearRegression()

# Fit the model
model.fit(X, y)

# Make predictions
predictions = model.predict(X)

# Evaluate the model
mse = np.mean((predictions - y)**2)
print('MSE:', mse)
```

2. Classification Models:

Decision Trees

Scikit-learn includes a wide range of classification models for predicting categorical target variables. Logistic regression, decision trees, random forests, and support vector machines (SVM) are some of the classification models available. Here's an example of training a decision tree classifier in scikit-learn:

```
In [ ]: from sklearn.tree import DecisionTreeClassifier
```

Create a decision tree classifier


```
In [ ]: model = DecisionTreeClassifier()
```

Train the model

```
In [ ]: model.fit(X_train, y_train)
```

Make predictions

```
In [ ]: y_pred = model.predict(X_test)
```

3. Model Selection and Evaluation using Logistic Regression:

Scikit-learn provides tools for model selection and evaluation, including techniques for cross-validation, hyperparameter tuning, and model evaluation metrics. Here's an example of performing cross-validation and evaluating a model's performance:

```
In [ ]: from sklearn.model_selection import cross_val_score
        from sklearn.linear_model import LogisticRegression
```

Create a logistic regression model

```
In [ ]: model = LogisticRegression()
```

Perform cross-validation

```
In [ ]: scores = cross_val_score(model, X, y, cv=5)
```

Calculate the mean accuracy

```
In [ ]: mean_accuracy = scores.mean()
```

4. Random Forest:

```
In [ ]: from sklearn.ensemble import RandomForestClassifier
```

Create a Random Forest classifier

```
In [ ]: model = RandomForestClassifier()
```

Train the model

```
In [ ]: model.fit(X_train, y_train)
```

Make predictions

```
In [ ]: y_pred = model.predict(X_test)
```

5. Ensemble Methods:

Scikit-learn provides ensemble methods that combine multiple base models to improve overall performance. Random Forests, AdaBoost, and Gradient Boosting are popular ensemble methods in scikit-learn. Here's an example of training a Random Forest classifier:

Semi-Supervised Learning

Semi-supervised learning is a type of machine learning where a model is trained on both labeled and unlabeled data. It combines elements of supervised learning (using labeled data) and unsupervised learning (using unlabeled data) to improve the model's performance.

Semi-supervised learning is particularly useful when labeled data is scarce but unlabeled data is abundant. It allows the model to leverage the unlabeled data to improve its performance.

In this description, we'll explore how to perform semi-supervised learning using Python. We'll use the scikit-learn library, which provides various machine learning algorithms and tools.

First, let's start by installing scikit-learn if you haven't already. You can use the following command to install it via pip:

```
In [ ]: pip install scikit-learn
```

```
In [ ]: # Step 1: Import the necessary libraries
from sklearn.semi_supervised import LabelPropagation
from sklearn.datasets import make_classification

# Step 2: Generate a synthetic dataset
X, y = make_classification(n_samples=1000, n_features=10, n_informative=8, n_redundant=2, random_state=42)

# Step 3: Split the dataset into labeled and unlabeled data
labeled_X = X[:100] # First 100 samples are labeled
labeled_y = y[:100] # Corresponding labels for the labeled samples
unlabeled_X = X[100:] # Remaining samples are unlabeled

# Step 4: Create a semi-supervised learning model and fit it
model = LabelPropagation()
model.fit(labeled_X, labeled_y)

# Step 5: Predict the labels for the unlabeled data
predicted_labels = model.transduction_[100:]

# Step 6: Evaluate the model's performance on the labeled data
accuracy = model.score(labeled_X, labeled_y)

# Step 7: Print the results
print("Accuracy on labeled data: {:.2f}%".format(accuracy * 100))
```

Confusion Matrix:

Confusion matrix is a popular tool for evaluating the performance of classification models. It provides a comprehensive summary of how well the model has classified the different classes in a dataset. In this description, we'll explore how to create and interpret a confusion matrix using Python.

Analyzing the confusion matrix can provide insights into the performance of the classification model, such as identifying which classes are frequently misclassified.

```
In [ ]: # Step 1: Import the necessary libraries
from sklearn.metrics import confusion_matrix
```

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

# Step 2: Define the true Labels and predicted Labels
true_labels = [1, 0, 1, 1, 0, 1, 0, 0, 1, 0]
predicted_labels = [1, 1, 1, 0, 0, 1, 0, 1, 0, 1]

# Step 3: Create a confusion matrix
cm = confusion_matrix(true_labels, predicted_labels)

# Step 4: Visualize the confusion matrix
plt.figure(figsize=(6, 4))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues")
plt.title("Confusion Matrix")
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.show()
```

6. Scikit Learn - Project on Label Propagation

Label Propagation digits active learning

Demonstrates an active learning technique to learn handwritten digits using label propagation.

We start by training a label propagation model with only 10 labeled points, then we select the top five most uncertain points to label. Next, we train with 15 labeled points (original 10 + 5 new ones). We repeat this process four times to have a model trained with 30 labeled examples. Note you can increase this to label more than 30 by changing `max_iterations`. Labeling more than 30 can be useful to get a sense for the speed of convergence of this active learning technique.

A plot will appear showing the top 5 most uncertain digits for each iteration of training. These may or may not contain mistakes, but we will train the next model with their true labels.

Imports

```
In [ ]: import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
```

```
from sklearn import datasets
from sklearn.semi_supervised import LabelSpreading
from sklearn.metrics import classification_report, confusion_matrix
```

Python Implementation

```
In [ ]: import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

from sklearn import datasets
from sklearn.semi_supervised import LabelSpreading
from sklearn.metrics import classification_report, confusion_matrix

digits = datasets.load_digits()
rng = np.random.RandomState(0)
indices = np.arange(len(digits.data))
rng.shuffle(indices)

X = digits.data[indices[:330]]
y = digits.target[indices[:330]]
images = digits.images[indices[:330]]

n_total_samples = len(y)
n_labeled_points = 40
max_iterations = 5

unlabeled_indices = np.arange(n_total_samples)[n_labeled_points:]
f = plt.figure()

for i in range(max_iterations):
    if len(unlabeled_indices) == 0:
        print("No unlabeled items left to label.")
        break
    y_train = np.copy(y)
    y_train[unlabeled_indices] = -1

    lp_model = LabelSpreading(gamma=0.25, max_iter=20)
    lp_model.fit(X, y_train)

    predicted_labels = lp_model.transduction_[unlabeled_indices]
```

```

true_labels = y[unlabeled_indices]

cm = confusion_matrix(true_labels, predicted_labels, labels=lp_model.classes_)

print("Iteration %i %s" % (i, 70 * "_"))
print(
    "Label Spreading model: %d labeled & %d unlabeled (%d total)"
    % (n_labeled_points, n_total_samples - n_labeled_points, n_total_samples)
)

print(classification_report(true_labels, predicted_labels))

print("Confusion matrix")
print(cm)

# compute the entropies of transduced label distributions
pred_entropies = stats.distributions.entropy(lp_model.label_distributions_.T)

# select up to 5 digit examples that the classifier is most uncertain about
uncertainty_index = np.argsort(pred_entropies)[::-1]
uncertainty_index = uncertainty_index[
    np.in1d(uncertainty_index, unlabeled_indices)
][:5]

# keep track of indices that we get labels for
delete_indices = np.array([], dtype=int)

# for more than 5 iterations, visualize the gain only on the first 5
if i < 5:
    f.text(
        0.05,
        (1 - (i + 1) * 0.183),
        "model %d\n\nfit with\n%d labels" % ((i + 1), i * 5 + 10),
        size=10,
    )
for index, image_index in enumerate(uncertainty_index):
    image = images[image_index]

    # for more than 5 iterations, visualize the gain only on the first 5
    if i < 5:
        sub = f.add_subplot(5, 5, index + 1 + (5 * i))
        sub.imshow(image, cmap=plt.cm.gray_r, interpolation="none")
        sub.set_title(
            "predict: %i\ntrue: %i"

```

```

        % (lp_model.transduction_[image_index], y[image_index]),
        size=10,
    )
    sub.axis("off")

    # Labeling 5 points, remote from Labeled set
    (delete_index,) = np.where(unlabeled_indices == image_index)
    delete_indices = np.concatenate((delete_indices, delete_index))

    unlabeled_indices = np.delete(unlabeled_indices, delete_indices)
    n_labeled_points += len(uncertainty_index)

f.suptitle(
    "Active learning with Label Propagation.\nRows show 5 most "
    "uncertain labels to learn with the next model.",
    y=1.15,
)
plt.subplots_adjust(left=0.2, bottom=0.03, right=0.9, top=0.9, wspace=0.2, hspace=0.85)
plt.show()

```

Iteration 0

Label Spreading model: 40 labeled & 290 unlabeled (330 total)

	precision	recall	f1-score	support
0	1.00	1.00	1.00	22
1	0.78	0.69	0.73	26
2	0.93	0.93	0.93	29
3	1.00	0.89	0.94	27
4	0.92	0.96	0.94	23
5	0.96	0.70	0.81	33
6	0.97	0.97	0.97	35
7	0.94	0.91	0.92	33
8	0.62	0.89	0.74	28
9	0.73	0.79	0.76	34
accuracy			0.87	290
macro avg	0.89	0.87	0.87	290
weighted avg	0.88	0.87	0.87	290

Confusion matrix

```

[[22  0  0  0  0  0  0  0  0  0]
 [ 0 18  2  0  0  0  1  0  5  0]
 [ 0  0 27  0  0  0  0  0  2  0]
 [ 0  0  0 24  0  0  0  0  3  0]
 [ 0  1  0  0 22  0  0  0  0  0]
 [ 0  0  0  0  0 23  0  0  0 10]

```

```
[ 0 1 0 0 0 0 34 0 0 0]
[ 0 0 0 0 0 0 0 30 3 0]
[ 0 3 0 0 0 0 0 0 25 0]
[ 0 0 0 0 2 1 0 2 2 27]]
```

Iteration 1

Label Spreading model: 45 labeled & 285 unlabeled (330 total)

	precision	recall	f1-score	support
0	1.00	1.00	1.00	22
1	0.79	1.00	0.88	22
2	1.00	0.93	0.96	29
3	1.00	1.00	1.00	26
4	0.92	0.96	0.94	23
5	0.96	0.70	0.81	33
6	1.00	0.97	0.99	35
7	0.94	0.91	0.92	33
8	0.77	0.86	0.81	28
9	0.73	0.79	0.76	34
accuracy			0.90	285
macro avg	0.91	0.91	0.91	285
weighted avg	0.91	0.90	0.90	285

Confusion matrix

```
[[22 0 0 0 0 0 0 0 0 0]
 [ 0 22 0 0 0 0 0 0 0 0]
 [ 0 0 27 0 0 0 0 0 2 0]
 [ 0 0 0 26 0 0 0 0 0 0]
 [ 0 1 0 0 22 0 0 0 0 0]
 [ 0 0 0 0 0 23 0 0 0 10]
 [ 0 1 0 0 0 0 34 0 0 0]
 [ 0 0 0 0 0 0 0 30 3 0]
 [ 0 4 0 0 0 0 0 0 24 0]
 [ 0 0 0 0 2 1 0 2 2 27]]
```

Iteration 2

Label Spreading model: 50 labeled & 280 unlabeled (330 total)

	precision	recall	f1-score	support
0	1.00	1.00	1.00	22
1	0.85	1.00	0.92	22
2	1.00	1.00	1.00	28
3	1.00	1.00	1.00	26
4	0.87	1.00	0.93	20
5	0.96	0.70	0.81	33
6	1.00	0.97	0.99	35
7	0.94	1.00	0.97	32
8	0.92	0.86	0.89	28

	9	0.73	0.79	0.76	34
accuracy				0.92	280
macro avg	0.93	0.93	0.93		280
weighted avg	0.93	0.92	0.92		280

Confusion matrix

```
[[22  0  0  0  0  0  0  0  0  0]
 [ 0 22  0  0  0  0  0  0  0  0]
 [ 0  0 28  0  0  0  0  0  0  0]
 [ 0  0  0 26  0  0  0  0  0  0]
 [ 0  0  0  0 20  0  0  0  0  0]
 [ 0  0  0  0  0 23  0  0  0 10]
 [ 0  1  0  0  0  0 34  0  0  0]
 [ 0  0  0  0  0  0  0 32  0  0]
 [ 0  3  0  0  1  0  0  0 24  0]
 [ 0  0  0  0  2  1  0  2  2 27]]
```

Iteration 3

Label Spreading model: 55 labeled & 275 unlabeled (330 total)

	precision	recall	f1-score	support
0	1.00	1.00	1.00	22
1	0.85	1.00	0.92	22
2	1.00	1.00	1.00	27
3	1.00	1.00	1.00	26
4	0.87	1.00	0.93	20
5	0.96	0.87	0.92	31
6	1.00	0.97	0.99	35
7	1.00	1.00	1.00	31
8	0.92	0.86	0.89	28
9	0.88	0.85	0.86	33

accuracy			0.95	275
macro avg	0.95	0.95	0.95	275
weighted avg	0.95	0.95	0.95	275

Confusion matrix

```
[[22  0  0  0  0  0  0  0  0  0]
 [ 0 22  0  0  0  0  0  0  0  0]
 [ 0  0 27  0  0  0  0  0  0  0]
 [ 0  0  0 26  0  0  0  0  0  0]
 [ 0  0  0  0 20  0  0  0  0  0]
 [ 0  0  0  0  0 27  0  0  0  4]
 [ 0  1  0  0  0  0 34  0  0  0]
 [ 0  0  0  0  0  0  0 31  0  0]
 [ 0  3  0  0  1  0  0  0 24  0]
 [ 0  0  0  0  2  1  0  0  2 28]]
```

Iteration 4


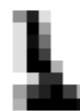
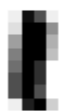













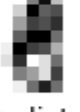



Label Spreading model: 60 labeled & 270 unlabeled (330 total)

	precision	recall	f1-score	support
0	1.00	1.00	1.00	22
1	0.96	1.00	0.98	22
2	1.00	0.96	0.98	27
3	0.96	1.00	0.98	25
4	0.86	1.00	0.93	19
5	0.96	0.87	0.92	31
6	1.00	0.97	0.99	35
7	1.00	1.00	1.00	31
8	0.92	0.96	0.94	25
9	0.88	0.85	0.86	33
accuracy			0.96	270
macro avg	0.95	0.96	0.96	270
weighted avg	0.96	0.96	0.96	270

Confusion matrix

```
[[22  0  0  0  0  0  0  0  0  0]
 [ 0 22  0  0  0  0  0  0  0  0]
 [ 0  0 26  1  0  0  0  0  0  0]
 [ 0  0  0 25  0  0  0  0  0  0]
 [ 0  0  0  0 19  0  0  0  0  0]
 [ 0  0  0  0  0 27  0  0  0  4]
 [ 0  1  0  0  0  0 34  0  0  0]
 [ 0  0  0  0  0  0  0 31  0  0]
 [ 0  0  0  0  1  0  0  0 24  0]
 [ 0  0  0  0  2  1  0  0  2 28]]
```

Active learning with Label Propagation.
 Rows show 5 most uncertain labels to learn with the next model.

model 1	predict: 1 true: 1	predict: 2 true: 1	predict: 1 true: 1	predict: 1 true: 1	predict: 3 true: 3
fit with 10 labels					
model 2	predict: 4 true: 4	predict: 4 true: 4	predict: 4 true: 4	predict: 8 true: 2	predict: 8 true: 7
fit with 15 labels					
model 3	predict: 2 true: 2	predict: 9 true: 5	predict: 9 true: 5	predict: 5 true: 9	predict: 7 true: 7
fit with 20 labels					
model 4	predict: 8 true: 8	predict: 1 true: 8	predict: 3 true: 3	predict: 4 true: 4	predict: 8 true: 8
fit with 25 labels					
model 5	predict: 1 true: 1	predict: 1 true: 1	predict: 7 true: 7	predict: 7 true: 7	predict: 1 true: 1
fit with 30 labels	