Implementation of SDR Classifier in Microsoft Azure Cloud

Chetan Chadha
Chetan.chadha@stud.fra-uas.de

Jagdeep Singh
Jagdeep.singh@stud.fra-uas.de

Maria Majid
Maria.majid@stud.fra-uas.de

Sidra Hussain
Sidra.hussain@stud.fra-uas.de

Sanjana Sharma
Sanjana.sharma@stud.fra-uas.de

Vishakha Babulal
Vishakha.babulal@stud.fra-uas.de

***Abstract*— The availability and agility of Cloud has made it beneficial for the growing current technology. Evidently, even the future technology will be completely dependent on Cloud computing. Exploiting this fact, the aim of this project is to get familiar with the services of Microsoft Azure environment. To illustrate cloud computing, the project SDR Classifier Implementation will be deployed in Microsoft Azure Cloud.**

Keywords— ***Cloud computing, Microsoft azure cloud, Docker container, Machine Learning SDR Classifier***

I. INTRODUCTION

Cloud computing is most likely to be the future of IT industry as it offers time and cost-effective solution to provide accessibility, agility, scalability, and reliability in development and deployment of Applications. Cloud is termed as the upcoming high-tech dawn in IT and works as renting resources or storage, provided by the Cloud service provider. Microsoft Azure is one of the cloud service providers. The main objective of this project is to get familiar with the Microsoft Azure environment, storages, and services in order to deploy and run the Software Engineering project on Cloud. The software engineering project was to implement Machine learning based SDR Classifier. The same code is used in this project to be deployed on Microsoft Azure Cloud.

Following is the brief description of the Machine learning based project "SDR classifier". Later in this section, the requirement of different Microsoft Azure Storages for this project will be explained.

a) Brief description of SDR classifier: The SDR Classifier is a simple one-layer feed forward classification neural network. It is an essential element in HTM (Hierarchical temporal memory) framework as it is responsible to detect and learn the relationship between the Temporal Memory's present state at time t and the future value at t+n, where n indicates no. of stages in future to be inferred [1]. SDR works by giving outliers in prediction probability by reinforcing the imperative weight matrix at each turn. The incorrect predictions are ensured to be penalized by successive iterations. The weight matrix, which is the key component in the classifier, is essentially updated as the classifier learns on every predicted value from temporal memory. In HTM-SDR Structure, a special form of input data is required to be processed, which is called Sparse Distributed Representation or SDR which is a large array of 0 and 1 bits. SDR Classifier performs advanced functionalities and cognitive tasks same as the human brain. This time-based prediction framework is designed to perform future data prediction based on learning and memorizing the data previously fed into it.

2) Requirements for deployment of ML project to Cloud: To run the ML based SDR classifier in Microsoft Azure Cloud, the knowldege of following elements are required.

a) Containerization of an App using Docker: Containerization is a time reducing and cost effective way to achieve rapid App deployment, as no hardware configurations and software installations are required to host a deployment. The principle purpose of containerization is to let multiple apps run in a single hardware inside their isolated containers, without interfering with files, resources, memory and processes utilized by other running apps. To fulfil these reqiurements, Docker serves as best solution, through which an app can be quickly deployed and run in its designed environment, both locally and on cloud. The app first needs to be package up as Docker image, which can be test locally by using Docker for Windows, or on clould by using Microsoft Azure Container Registry and Instance service [2].

b) Azure Storage: Microsoft Azure storage is cloud storage, which is highly secure, resilient, scalable, accessable and available anywhere at everytime. It has various storage abstractions including Blob storage, Table storage, Files storage & Queues Storage [3].

c) Azure Blob storage: Blob Storage serves as Object storage for Cloud. It is used for the storage of unstructured data. The main tasks of blob storage are to provide images and documents directly to the browser, store files for distributed access, generating log files, store data for backup and recovery etc. There are three resources types offered by Blob storage that includes storage account,

container and blob. Fig. 1.1 represents the Hierarchy of Blob storage [4].

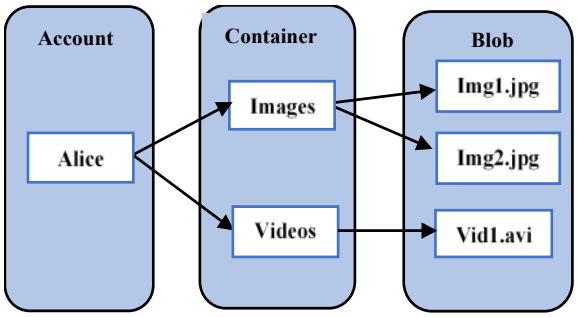

Fig. 1.1. Hierarchy of Blob storage

d) Azure Table storage: Azure Table storage, or Cosmos DB Table API serves as storage for structured data in cloud with schema less design. This makes it easier to adapt the data as per the requirement of perticular application. Table storage can be utilized for storing felxible datasheets and other types of metadata as per the service requirements. Table storage is consists of Storage account, storage table and entities, as shown in Fig. 1.2 [5].

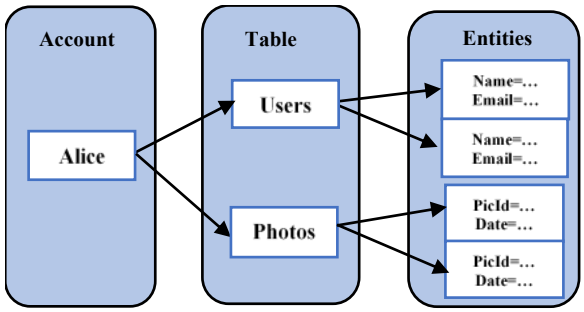

Fig. 1.2. Hierarchy of Table storage

e) Azure Queue Storage: Azure Queue Storage offers the cloud messaging service between the application components. It also provides support for task management and process workflow management. Queue storage service stores messages that can be accessable from all over the world via HTTP or HTTPS calls. The size of one Queue message can be up to 64B. There can be multiple messages present in a queue at one time up to the maximum storage capacity. It is also possible to create work Backlog using Queue storage [6].

In this paper, section II illustrates the general workflow of this project and the steps involved in the deployment of SDR classifier application on Azure Cloud. Section III describes the implemented code. Section IV explains final output after the deployment of SDR classifier application on Azure Cloud.

II. General Workflow

The main goal of this project is to run the already implemented Machine learning code, which is SDR classifier, on Cloud using Microsoft Azure cloud. The project workflow is illustrated in Fig. 1.3.

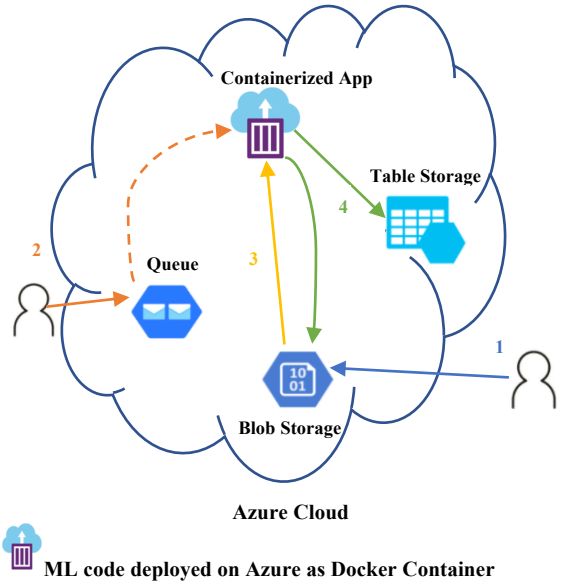

Fig. 1.3. Project's Workflow

Following are the implemented steps according to the above illustrated workflow, in order to achieve the goal of this project.

1) Package the ML project to Docker Container: First of all, the SDR classifier code is packaged up as Docker Image in docker container. The image build process is then automated. This docker image is uploaded to Azure Container Registry and the application then can be deployed using the Azure Container Instance service.

2) Upload the Input file to blob storage: The training input data files are uploaded as zip files to Azure Blob storage to start the application.

3) Send Trigger message in Queue: A specific message is sent in the Queue to trigger the training process.

4) Perform Deserialization: When the message from Queue is received, deserialization is performed and the code got triggered. Deserialization process is required here, as it converts and alters the data organization into a linear format for storage purpose.

5) Download Input file from blob storage: As soon as the code triggers, it will start downloading the training input data files from Azure Blob storage to extract the trainng data and start the SDR classifier experiment.

6) Run *the testcases for SDR classifier using the values from Input file:* All the test cases related to the SDR

classifier experiment will use the input values from the downloaded input data files and then start running.

7) Upload Result/Output file to blob and table storage: The output or results of the SDR classfier experiment and testcases will be added in an Output file which will be uploaded to both Azure Blob and Table storage.

8) Delete message from Queue: In the end, the message which was sent earlier in the Queue to trigger the training process will be deleted.

III. Code Description

The execution of this project according to the workflow illustrated above is implemented by the class Experiment.cs. It is situated in MyExperiment directory in the project folder and called by the main Program.cs for implementation.

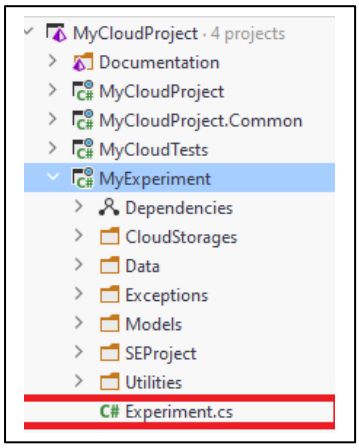

Fig. 3.1 Reference of Experiment.cs

The purpose of experiment class is to establish the folder path locally where files will be downloaded from blob storage and the data present in the downloaded file is executed and uploaded back on azure (Blob & Table). The program will then be executed until signaled to cancel via a cancellation token. The results of the program will then be uploaded in the storage blob and table. This stream of actions is illustrated in several methods discussed below.

```
public class Experiment : IExperiment
{
    5 usages
    public static string DataFolder { get; private set; }
    private IStorageProvider storageProvider;
    private ILogger logger;
    private MyConfig config;
    3 usages   chetan2510 +1 *
    public Experiment(IConfigurationSection configSection,
        IStorageProvider storageProvider, ILogger log)
    {
        this.storageProvider = storageProvider;
        logger = log;
        config = new MyConfig();
        configSection.Bind(config);

        //  Creates the directory where the input-data
        // from the blob will be stored
        DataFolder = Path.Combine(Environment.
                GetFolderPath(Environment.SpecialFolder
                    .LocalApplicationData),
            config.LocalPath);
        Directory.CreateDirectory(DataFolder);
```

Fig. 3.2 Experiment.cs constructor

The first three objects as shown in Fig 3.2 are established to be used throughout the code.

a) storageProvider: Concerns associations with the blob storage i.e. uploading files in blob and downloading file from it.
b) logger: Concerns appending structured messages regarding instant at hand throughout the code called logs.
c) config: Represents the configuration selected for the project, associates with container names, queue, and group ID etc.

After the object declarations, the constructor for the class is declared as shown in Fig 3.2. It is to be noted that `configSection` is the variable established to bind with `config` above. It is to be noted that the argument configSection make use of the descendent configuration (`IConfiguarationSection`) which internally uses `InitHelper` class, in order to read configuration or fetch details from the appconfig.json file of the project.

In the next step a directory is created, `Path.combine` analogy is used to concatenate the address using the two arguments that are `Environment.SpecialFolder` and locally available path as given in the configuration file, appconfig.json . This ensures a suitable file path.

```
public async Task RunQueueListener(CancellationToken cancelToken)
{
    CloudQueue queue = await AzureQueueOperations.
        CreateQueueAsync(config, logger);

    while (cancelToken.IsCancellationRequested == false)
    {
        var message = await queue.GetMessageAsync(cancelToken);
        try
        {
            if (message != null)
            {
                // STEP 1. Reading message from Queue and deserializing
                var experimentRequestMessage =
                    JsonConvert.DeserializeObject<ExerimentRequestMessage>
                        (message.AsString);
                logger?.LogInformation(
                    message: $"Received message from the queue with experimentID: "
                    +
                    $"{experimentRequestMessage.ExperimentId}, " +
                    $"description: {experimentRequestMessage.Description}, "
                    +
                    $"name: {experimentRequestMessage.Name}");
```

Fig. 3.3 Queue listener method

As described in section II, part 3, preceding the constructor is the method `RunQueueListener`. It aims at listening the queue message for the program to be executed as illustrated in Fig 3.3. Following describes how the method works:

The arguments of the method receive a token that should be cancelled in further time of execution with type `CancellationToken`. A queue is created at first using the storage account information of the user which contains

the configuration for the program to be executed. Following that, the state of the token is checked. At a cancel state (subject to the user pressing the key), the process is stopped, and an appropriate log is registered. At a non-cancel state, the method follows 6 steps.

Step I: The message in queue is read as illustrated in the above figure 3.4. For a not null message (which signals for the execution of the program), the message is deserialized or converted from byte stream to object in memory. A log is then logged on the console showcasing details in the queue message such as groupID, name, description, etc.

```
// STEP 2. Downloading the input file from the blob storage
var fileToDownload :string = experimentRequestMessage.InputFile;
var localStorageFilePath :string = await storageProvider. // IStorageProvide
    DownloadInputFile(fileToDownload); // Task<string>

logger?.LogInformation(
    message: $"File download successful. Downloaded file link:" +
    $" {localStorageFilePath}");

// STEP 3. Running SE experiment with inputs from the input file
var result = await Run(localStorageFilePath);
result.Description = experimentRequestMessage.Description;
var resultAsByte :byte[] = Encoding.UTF8.GetBytes( s: JsonConvert.
    SerializeObject(result));
```

Fig. 3.4 Download of input files and run of experiment

Step II: The name of the training data file present in the queue message is downloaded from the blob and the local path of the file returned is stored in the variable `localStorageFilePath`. As shown in figure 3.4 the method `DownloadInputFile` in class `AzureBlobOperation.cs` is used to download the input file. Its approach is straightforward as illustrated in below figure 3.5. The file name to be downloaded is passed in the arguments. First the local repository path is created, and the file name is appended. Next, using the storage account connection string an instance `blob` is initialized by using the credentials of the account and the URI of the file name in the storage blob. The contents in the blob are then downloaded in the file using the instance with `DownloadToFileAsync`.

```
public async Task<string> DownloadInputFile(string fileName)
{
    if (StringUtilities.isBlankOrNull(fileName))
    {
        throw new EmptyStringException(
            message: "File name cannot be empty or null");
    }

    string localStorageFilePath = Path.Combine(Experiment.DataFolde
        new FileInfo(fileName).Name);
    Microsoft.Azure.Storage.CloudStorageAccount
        cloudStorageAccount =
        Microsoft.Azure.Storage.CloudStorageAccount.
            Parse(config.StorageConnectionString);
    var blob = new CloudBlockBlob(new Uri(fileName),
        cloudStorageAccount.Credentials);
    await blob.DownloadToFileAsync(localStorageFilePath,
        FileMode.Create); // Task
    return localStorageFilePath;
}
```

Fig. 3.5 Download of input file method in AzureBlobStorage.cs

Step III: With the training data file in the local repository the program is run by providing the path of the file stored. The result of the program returned by `Run` is serialized or converted into byte stream for further processing. During the execution of the `Run` method as illustrated in the below Fig. 3.6, firstly it desterilizes the data from the downloaded file after that it runs the software engineering experiment using method `RunSoftwareEngineeringExperiment`(figure 3.7-3.8) and records the results with the time instants and duration of execution.

```
public async Task<ExperimentResult> Run(string localFileName)
{
    var seProjectInputDataList =
        JsonConvert.DeserializeObject<List<SeProjectInputDataModel>>
            (FileUtilities.ReadFile(localFileName));

    var startTime = DateTime.UtcNow;

    // running until the input ends
    string uploadedDataURI =
        await RunSoftwareEngineeringExperiment(seProjectInputDataList);

    // Added delay
    Thread.Sleep( millisecondsTimeout: 5000);
    var endTime = DateTime.UtcNow;

    logger?.LogInformation(
        message: $"Ran all test cases as per input from the blob storage");

    long duration = endTime.Subtract(startTime).Seconds;

    var res = new ExperimentResult( partitionKey: this.config.GroupId,
        rowKey: Guid.NewGuid().ToString());
    UpdateExperimentResult(res, startTime, endTime, duration,
        localFileName, uploadedDataURI);
    return res;
}
```

Fig. 3.6 Run method in Experiment.cs

In Fig 3.7, RunSoftwareEngineeringExperiment showcases the loop at which input data is processed by calling the different methods of the SDR classifier. Then, a file is created to record the respective results.

```
private async Task<string> RunSoftwareEngineeringExperiment(
    List<SeProjectInputDataModel> seProjectInputDataList)
{
    // For the set of inputs alpha does not changes
    var sdr = new SdrClassifier(seProjectInputDataList[0].Alpha);

    // Step 1: Run all test cases in a file
    foreach (var input in seProjectInputDataList)
    {
        var classification = new List<object> {input.BucketIndex,
         input.ActualValueInBucket};

        // After this step bucket values will be updated
        sdr.Compute(0, classification, input.InputFromTemporalMemory);
        var predictionResult = sdr.Predict(input.InputFromTemporalMemory);
        var seProjectOutputModel = new SeProjectOutputModel(predictionResult);
        var outputAsByte = Encoding.UTF8.GetBytes(JsonConvert.
        SerializeObject(seProjectOutputModel));

        // Writing output in a file
        FileUtilities.WriteDataInFile("CombinedOutputOfSoftwareExperiment.txt",
            predictionResult,
            input.BucketIndex);
```

Fig. 3.7 Running Test cases

Fig. 3.8 shows how the SDRClassifer output file is uploaded in blob using method UploadResultFile (discussed in step IV).

```
// Step 2: Uploading output to blob storage
var uploadedUri =
    await storageProvider.
    UploadResultFile("CombinedOutputOfSoftwareExperiment.txt",
        null);
logger?.LogInformation(
    $"Test cases output file uploaded successful. Blob URL:" +
    $" {Encoding.ASCII.GetString(uploadedUri)}");

// return a string and delete the combined file if possible
return Encoding.ASCII.GetString(uploadedUri);
```

Fig. 3.8 Uploading Test case file on blob

```
// STEP 4. Uploading result file to blob storage
var uploadedUri :byte[] =
    await storageProvider.UploadResultFile
    ("ResultFile-" + Guid.NewGuid() + ".txt",
        resultAsByte); // Task<byte[]>
logger?.LogInformation( message: $"Uploaded result file on blob")
result.SeExperimentOutputBlobUrl =
    Encoding.ASCII.GetString(uploadedUri);

// STEP 5. Uploading result file to table storage
await storageProvider.UploadExperimentResult(result);

// STEP 6. Deleting the message from the queue
await queue.DeleteMessageAsync(message, cancelToken);
```

Fig. 3.9 Upload file to blob, table and delete of message in queue

Step IV: The result of the program is uploaded in blob storage with the method `UploadResultFile` in class `AzureBlobOperations.cs`. An appropriate log is appended, and the blob URI is stored in result. The method `UploadResultFile` uses a straightforward approach as illustrated in Fig. 3.10, a blob service client object is created to create an object of container client. The file name of result is appended in an appropriate local path.

```
public async Task<byte[]> UploadResultFile(string fileName,
    byte[] data)
{

    if (StringUtilities.isBlankOrNull(fileName))
    {
        throw new EmptyStringException(
            message: "File name cannot be empty or null");
    }
    // Creates a BlobServiceClient object which
    // will be used to create a container client
    BlobServiceClient blobServiceClient =
        new BlobServiceClient(config.StorageConnectionString);

    // Create the container and return a container
    // client object
    BlobContainerClient containerClient =
        blobServiceClient.GetBlobContainerClient(config.ResultContainer);

    // Create a local file in the ./data/ directory for
    // uploading and downloading
    string localFilePath = Path.Combine(Experiment.
        DataFolder, fileName);
```

Fig. 3.10 Blob container object creation

The file is then uploaded to the blob storage and the URI of the file in blob is returned as shown in Fig. 3.11.

```
// Write text to the file
// Adding a check to write a data in a file
// only if data is not equal to null
// This is important as we need to re-use this
// method to upload a file in which data has already been written
if (data != null)
{
    File.WriteAllBytes(localFilePath, data);
}

// Get a reference to a blob
var blobClient = containerClient.GetBlobClient(fileName);

// Open the file and upload its data
FileStream uploadFileStream = File.OpenRead(localFilePath);
await blobClient.UploadAsync(uploadFileStream, overwrite: true);
uploadFileStream.Close();
return Encoding.ASCII.GetBytes( s: blobClient.Uri.ToString());
```

Fig. 3.11 Upload file in blob

Step V: The result is uploaded in storage table using `UploadExperimentResult` in the class `AzureBlobOperation.cs` illustrated Fig. 3.12.

```
public async Task UploadExperimentResult(ExperimentResult result)
{
    if (result == null)
    {
        throw new ObjectShouldNotBeNUllException(
            message: "Result object cannot be null");
    }

    CloudTable table =
        await AzureTableOperations.CreateTableAsync(config.ResultTable,
            config.StorageConnectionStringCosmosTable);
    await AzureTableOperations.InsertOrMergeEntityAsync(table, result);
}
```

Fig. 3.12 Upload file in table

A table is created using the storage connection string with method `CreateTableAsync` in the class `AzureTableOperations` and the results are either inserted if similar entity is not present in the table or merged if it is. That is done by `InsertOrMergeEntityAsync` method of class `AzureTableOperations.cs`

Step VI: Referring Fig 3.9, the message of the queue is deleted. The cancellation token and the message is passed in the method `DeleteMessageAsync`. This is to observe the cancellation token which the delete is taking place. If the cancel token signals to stop the process the delete will not occur.

IV. RESULTS

The extensive stream of methods discussed in this paper take much less time on accounts of running the experiment. It is due to this property of Microsoft Azure Environment why it is utilized in deploying SDR Classifier project. Following images illustrates the run of the code.

Figure 4.1, as soon as the experiment starts, a queue is created simultaneously.

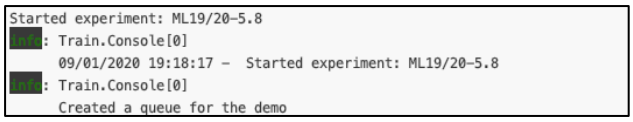

Fig 4.1 Queue creation

Figure 4.2, next, on receiving the message from the Azure queue, the console application starts downloading the input file.

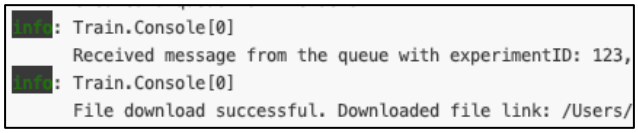

Fig 4.2 File download locally

Figure 4.3, then, the experiment is executed, and the result file named as "CombineOutputofSoftwareExperiment.txt" is uploaded on Blob storage and Table Storage, appropriate logs are displayed on the console.

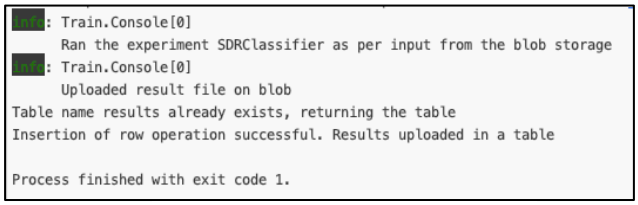

Fig 4.3 File uploaded in Table in Blob storage

REFERENCES

[1] "Implementation of SDR Classifier", [Online]. Available: https://github.com/UniversityOfAppliedSciencesFrankfurt/se-cloud-2019-2020/blob/GroupA2020/Source/HTM/GroupA2020DocumentationAndVideo/SDR_Classifier_Research_Paper.pdf

[2] "Deploy and run and containerized web app with Azure App Service", [Online]. Available: https://docs.microsoft.com/en-us/learn/modules/deploy-run-container-app-service/

[3] "Architect storage infrastructure in Azure" , [Online] . Available: https://docs.microsoft.com/en-us/learn/paths/architect-storage-infrastructure/

[4] "Azure Blob storage documentation", [Online]. Available: https://docs.microsoft.com/en-us/azure/storage/blobs/storage-blobs-overview

[5] "Azure Cosmos DB Documentation", [Online]. Available: https://docs.microsoft.com/en-us/azure/cosmos-db/tutorial-develop-table-dotnet

[6] "Azure Queue storage Documentation", [Online]. Available: https://docs.microsoft.com/en-us/azure/storage/queues/storage-dotnet-how-to-use-queues?tabs=dotnet