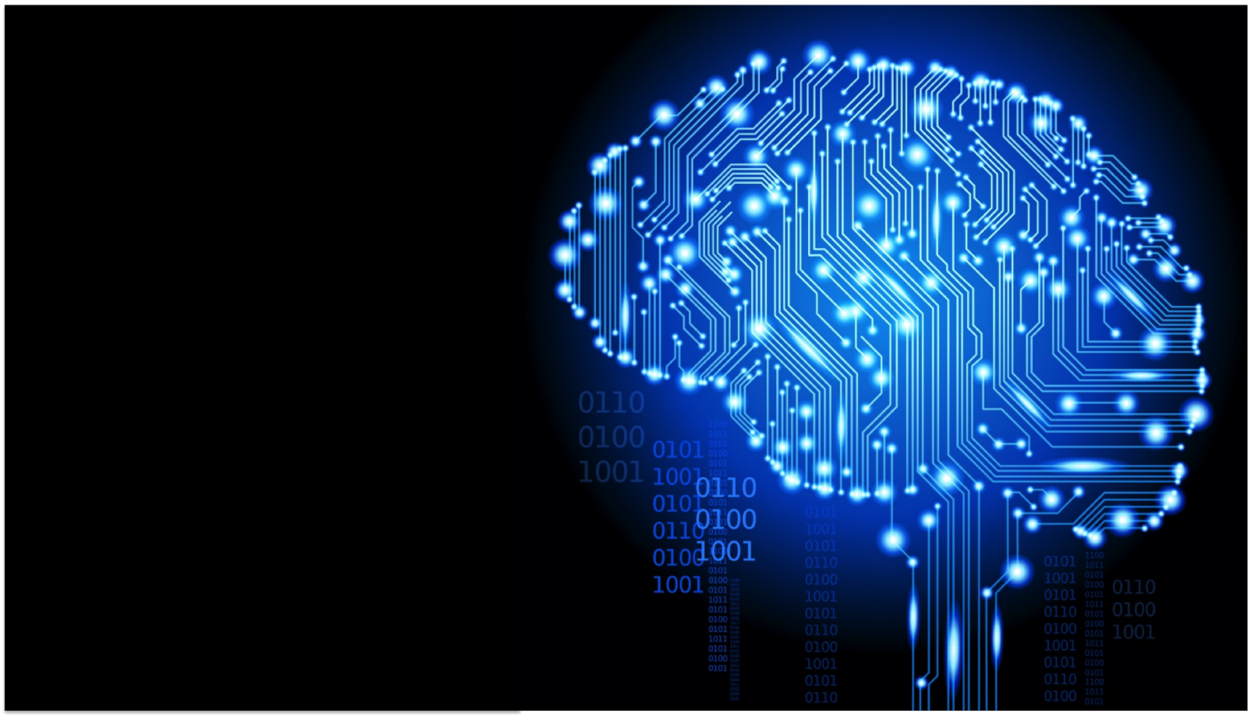
CSP 554 BIG DATA TECHNOLOGIES
Project Report



PROJECT GROUP

April 30, 2020

Group Members:

-Dharamvir Yadav, Group Voice

-Rishab Panyam

-Ramya Rajendra

-Marta Garcia Ruiz de Leon

ABSTRACT

Deep learning has been proved to produce highly accurate machine learning models in many fields. Medical imaging processing is one of these fields where the use of deep learning classification models is helping in the diagnosis of many diseases. One of these diseases is cancer. An early detection of malignant cells in a patient increases the chances of correct treatment and survival. Studies have shown that computer-aided diagnosis models have improved diagnosis accuracy [1]. However, when using imaging processing on a single node, many problems can arise. In order to overcome these problems, this project will focus on distributed deep learning. Deep learning pipelines include high-level APIs for scalable deep learning in Python with Apache Spark. By using Apache Spark, we will be able to reduce training time and run parallel experiments on many devices.

INTRODUCTION

Identification of malignant from normal cells from microscopic images is very difficult because morphologically both types of cells appear similar. They are also very expensive and not widely available. Usually cancer cells are detected in advanced stages using microscopic images because of the medical expertise as those malignance cells are present in a much greater number as compared to normal people. Therefore, it is very important to detect those cells at an early stage for better cure and improving the survival of the subject. This is where cell classification via image processing comes into play to provide a solution which can be deployed easily at lesser cost.

Image classification models are proving to be very helpful in cancer cell detection problems. However, imaging processing models require very large datasets in order to obtain good results, which means expensive computation is required. This is where Apache Sparks comes into play. As a framework for distributed computation, Spark will allow us to build deep learning models which are very computationally heavy. There are many ways to do Deep learning with Apache Spark. These methods go from distributed DL with Keras and Pyspark, to Tensorflow on Spark or bigDL. In this project we will discuss the different methods that exist nowadays to work with deep learning models in Spark and we will implement some of these methods to see how they vary from each other.

For this project we will store the data in an Amazon Web Series S3 bucket and we will use Spark in two different platforms, one will be in an EMR cluster and the second will be using Amazon Sagemaker. All of these technologies mentioned will be explained in detail in the project.

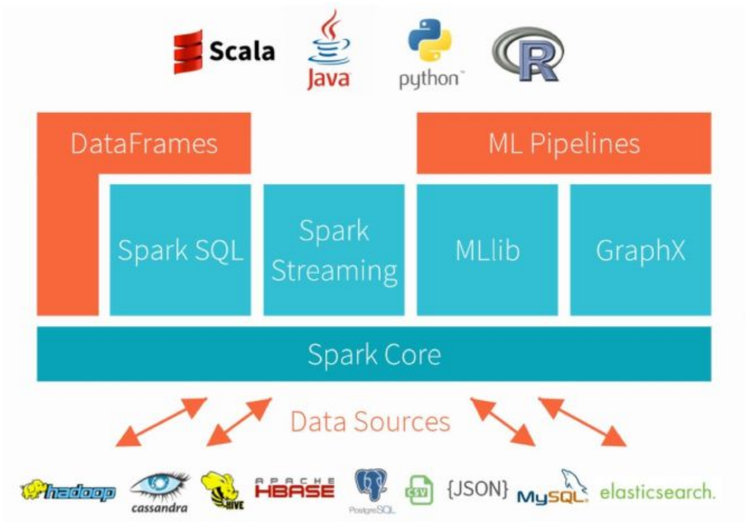
State of the Art: Documentation

In today's day and age data is created at a rate of aproximaltelly 2.5 quintillion bytes per day[2] and this rate will continue to increase as the amount of technology that is used increases. With local machines not able to process data and deploy models fast enough, distributed computing has become a necessity.

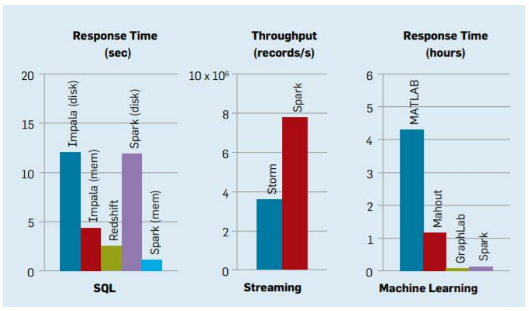
Distributed computing has resulted in the creation of various systems that can scale out these computations into different nodes. *"In the open source Apache Hadoop stack, systems like Storm1 and Impala9 are also specialized. Even in the relational database world, the trend has been to move away from "one-size-fits-all" systems. Unfortunately, most big data applications need to combine many different processing types. Specialized engines can thus create both complexity and inefficiency; users must stitch together disparate systems, and some applications simply cannot be expressed efficiently in any engine*."[3]. In this context Apache Spark was created in order to create a unified engine.

Apache Spark

Apache Spark, was a project started in the University of California, Berkeley in 2009 in which a unified engine for distributed data processing was created. "*Using Resilient Distributed Datasets, RDDs, Spark can capture a wide range of processing workloads that previously needed separate engines*".[4] Apache Spark has got four main libraries that run over the common engine: SparkSQL , Spark Streaming, MLlib and Graph X. Since Spark uses a unified API, applications are more easy to develop. Moreover, Spark can run various functions over the same data. One of the main components of Spark is that all the transformations are lazy,



meaning that they do not compute all the results right way. Instead the transformations are only computed when an action requires a result to be given to the driver program. Spark exposes RDD's through a functional programming in Scala, Java, Python and R, where the user can use local functions to run on the cluster.



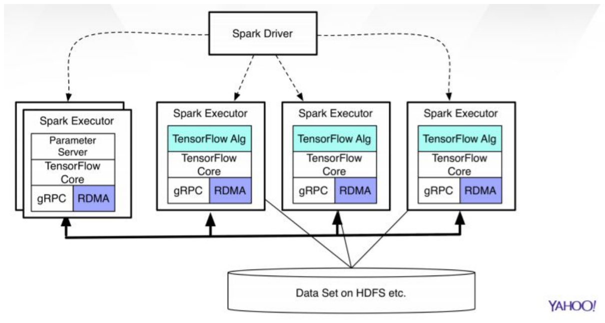
The key of Spark is that even though all of these libraries run on the same engine it has got as good a response as specialized engines.

In this project, we will use Spark in a new context that is Deep learning.

Deep Learning Landscape Frameworks

Deep learning like it was mentioned before, requires very big datasets and models that are computationally very heavy. In recent years, there has been an emergence of solutions that combine Spark and Deep learning. The most popular frameworks will be discussed and as an objective of this project one will be chosen to be implemented.

TensorflowOnSpark:



It was developed by Yahoo for large-scale distributed deep learning on a Hadoop cluster in Yahoo's private cloud. It supports all of TensorFlow's functionalities and it's integrated with existing data and pipelines. It is an open source system that scales up with minimum changes in the code.[5]

CaffeonSpark

CaffeOnSpark is another Yahoo application of deep learning in Spark clusters. This framework is complementary to other non-deep learning libraries suchs as Spark SQL and

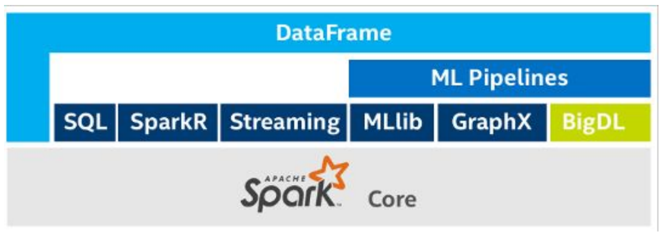
MLlib. This application supports neural network model training, testing and feature extraction and can be deployed in a private or public cloud.[6]

Elephas: Distributed DL with Keras and PySpark

Elephas is an extension of Keras that allows the user to run distributed deep learning models at scale with Spark. Elphas supports the following applications: from parallel training of models to hyper-parameter optimization or distributed training of ensemble models by implementing algorithms on top of Keras using Sparks RDDs and data frames.[7]

BigDL

BidDL is another distributed deep learning library for Apache Spark. Modeled after Torch, BigDL provides support for deep learning including numeric computing via Tensor and high level neural networks. Moreover, users can load pretrained models from Caffe or Torch. It also uses Intel's Kernel library and parallel computing techniques to achieve very high performance.[8]



Spark DL: Deep Learning Pipelines

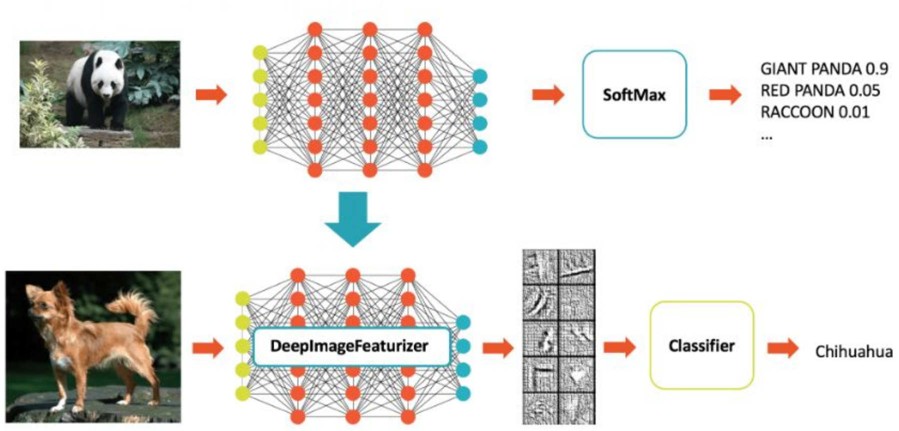
Deep Learning Pipelines is an open source library created by Databricks that provides high level APIs for scalable deep learning in Python with Apache Spark.[9] Built by the creators of Apache Spark, it is prepared to be merged into the official API. One of the main advantages of SparkDL is that it is very easy to implement, and it uses Tensorflow and Keras as its backend. This library builds on Apache Spark ML pipelines for the training of models and with Spark Dataframes and SQL for deploying the models[10]. It allows the user to apply pre-trained models as transformers in a Spark ML pipeline. It allows transfer learning as well as distributed hyperparameter tuning and the deployment of models in Data frames and SQL.

As we can see there are many frameworks to work with deep learning in Spark. For this project we will try and implement SparkDL and in parallel we will implement the model in

Amazon SageMaker. For the building of the models, transfer learning (discussed below) will be applied.

Transfer Learning

One of the quickest ways to use deep learning is through transfer learning. Transfer learning is based on using pre-trained models from libraries like Keras and Tensorflow. These models have been trained in a particular dataset, so when applied to a new dataset they are not optimized. However, optimizing to the new dataset is much quicker than building up the model from scratch. Transfer learning is being used specifically in many cases like the one proposed for this project, cancer detection.[11]



AWS

For this project, we will be using Amazon Web Services to create our Spark clusters and storing our data.

S3 buckets

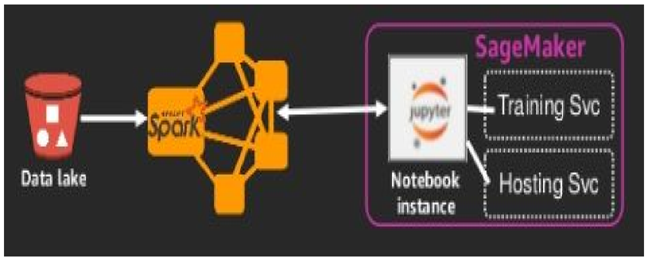
The data for this project will be stored in Amazon S3 buckets. S3 is an object storage service in which customers can store any amount of data. For this project the whole image dataset will be stored in S3 buckets.

EMR Clusters and Jupyter Notebook

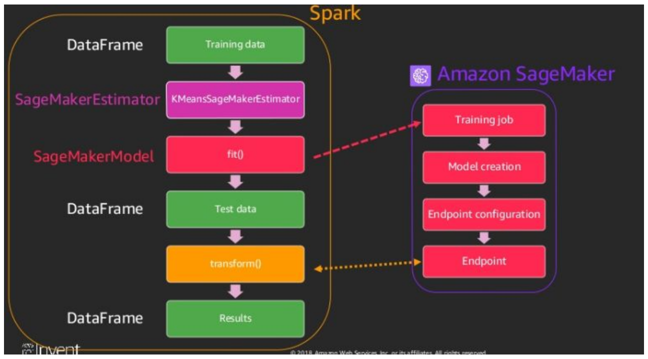
EMR is a cloud big data platform that allows us to use many tools like Apache Spark, Apache Hive, Apache HBase, Apache Hudi or Presto. It is a very easy to use application in which EMR notebooks can be deployed allowing users to collaborate in projects easily. For this project we will use an Amazon EMR Notebook, which is based on Jupyter Notebooks and it's designed for Apache Spark. It allows using languages like PySpark, SparkSQL, Spark R and Scala. [12]

Amazon Sagemaker

Another technology that will be used for this project is Amazon SageMaker. It is a fully managed machine learning service that allows users to build and train machine and deep learning models. It provides an integrated Jupyter authoring notebook instance so that servers don't have to be managed. It also provides common machine learning algorithms that are optimized to run in an efficient way in a distributed environment.



Amazon Sagemaker allows Spark on an Amazon EMR. The SageMaker-Spark exists to support Spark and Sagemaker integration in both directions. You can run Spark on an Amazon EMR and connect it to the SageMaker notebook where you can train models.[13]



Description of Experiments

Objectives

The objective of this project is to classify leukemic B-lymphoblast cells (cancer cells) from normal B-lymphoid precursors (healthy cells). In order to do so, we will build a deep learning classifier model.

As we know, deep learning problems require a high computation power. Thus, the key of this project is to develop a deep learning image classifier in a distributed system. In order to do so, two approaches will be taken:

1. Building a deep learning model in Amazon SageMaker: We will build the image classifier in SageMaker using a Spark instance in the notebook. The model will be built using transfer learning and then it will be optimized for our data.
2. Build a deep learning model in a EMR Notebook with a Spark instance using a deep learning framework: We will build an image classifier to identify if a patient has cancer and needs treatment or if he is healthy using Tensorflow on a Spark instance created in an EMR notebook.

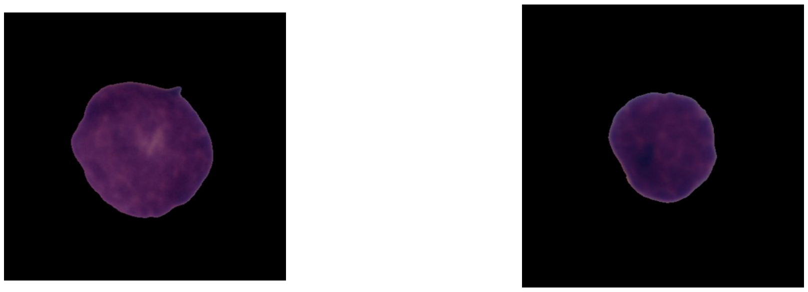
In order to validate the results of our project we will test our models on unseen data and calculate the F1 score as our metric.

Data Set Description

The dataset that will be used for this project was collected from a CodaLab[4] competition for classification of leukemic cells from normal cells in microscopic images.

The dataset contains images of leukemic B-lymphoblast cells (malignant cells) and normal B-lymphoid cells. The data set has been preprocessed, as cells have been normalized and segmented from the original images. The images have a final size of roughly 300x300 pixels.[14]

The data is divided in two folders, one for training the model and one for testing. The complete data set is composed of images from 118 patients. In each folder there are cell images from each patient.



Example of healthy(left) and malignant(right) cells from the data set.

All the images names follow the following standard naming convention:

UID_P_N_C_diagnosis

- UID_**P**: P is the subject ID
- UID_P_**N**: N is the number of image
- UID_P_N_**C**: C represents the cell count
- UID_P_N_**diagnosis**: **all** means cancer cell, **hem** means healthy cell.

Training Test set:

The dataset contains a total of 73 patients of which 47 have cancer and 26 are healthy. The separation of images in training and testing will be done by patients instead of by images of malignant and healthy cells. By doing so, we will not mix images from the same patient in the training and testing.

Methodology

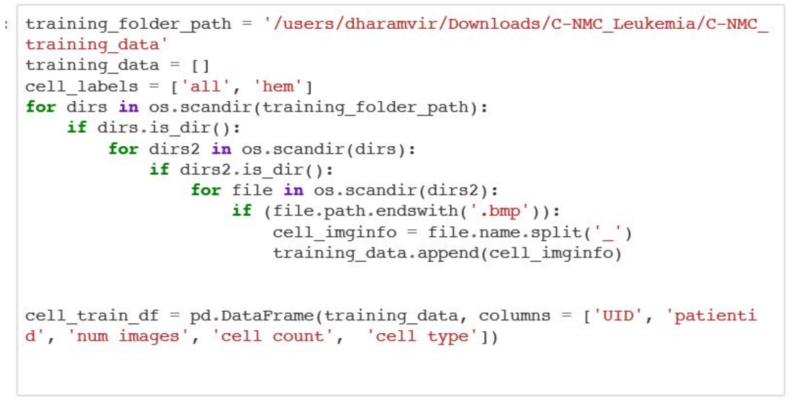
The methodology followed in two experiments that were conducted will be explained below, with the problems encountered and how they were resolved.

A. Sagemaker Experiment:

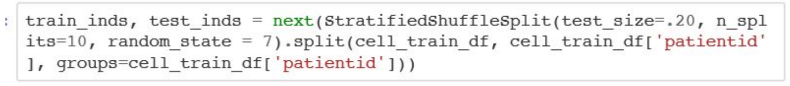
For this experiment, as it was mentioned before, the main objective was to create a deep learning classification model in a Spark instance using SageMaker. Transfer learning will be used instead of creating our own model from scratch. Therefore, we will need to do hyperparameter tuning to customize it to our data set of cancer cells.

1. Creating training and validation sets:

The dataset used for this project contained images of both cancer and healthy cells from different patients. We first create a training and validation set from the original data by separating our images by patient id. This way 80% of the patients are stored in the training set and 20% in the test set.



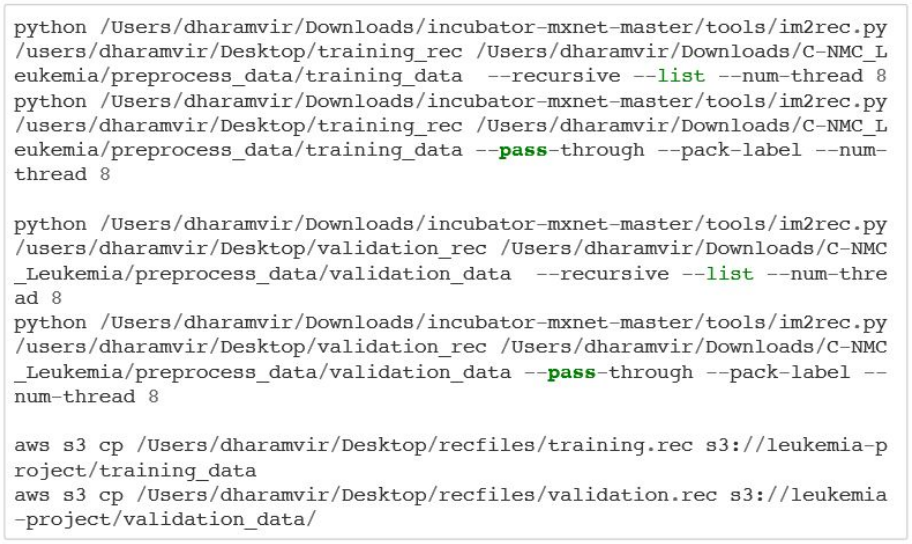
```
training_folder_path = '/users/dharamvir/Downloads/C-NMC_Leukemia/C-NMC_
training_data'
training_data = []
cell_labels = ['all', 'hem']
for dirs in os.scandir(training_folder_path):
    if dirs.is_dir():
        for dirs2 in os.scandir(dirs):
            if dirs2.is_dir():
                for file in os.scandir(dirs2):
                    if (file.path.endswith('.bmp')):
                        cell_imginfo = file.name.split('_')
                        training_data.append(cell_imginfo)


cell_train_df = pd.DataFrame(training_data, columns = ['UID', 'patienti
d', 'num images', 'cell count',  'cell type'])
```



```
train_inds, test_inds = next(StratifiedShuffleSplit(test_size=.20, n_spl
its=10, random_state = 7).split(cell_train_df, cell_train_df['patientid'
], groups=cell_train_df['patientid']))
```

Once the images have been downloaded, they are stored in a RECORDIO format, that is the format needed for the majority of transfer learning models in SageMaker.

After getting the images in the right format we upload them into the S3 bucket for the project called Leukemia project.

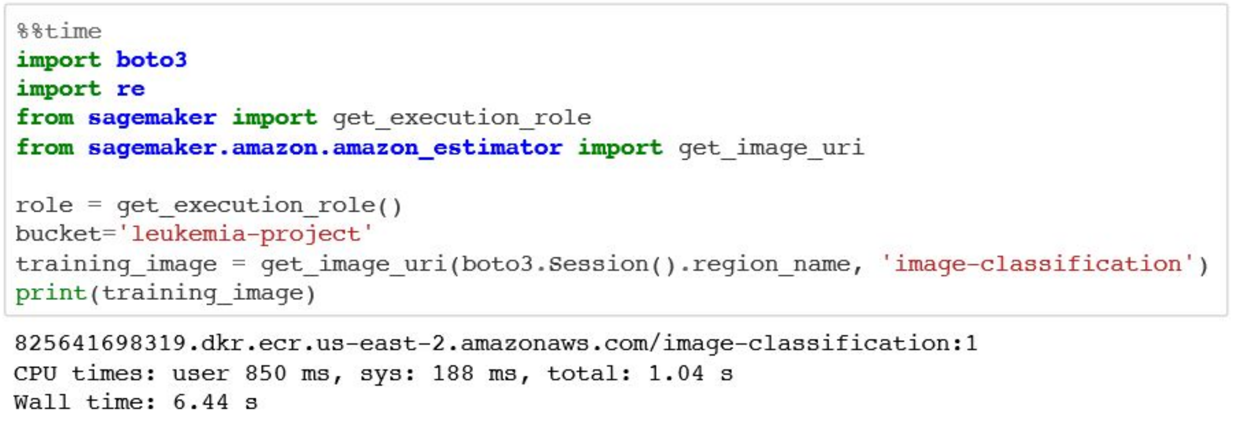


```
python /Users/dharamvir/Downloads/incubator-mxnet-master/tools/im2rec.py
/users/dharamvir/Desktop/training_rec /Users/dharamvir/Downloads/C-NMC_L
eukemia/preprocess_data/training_data  --recursive --list --num-thread 8
python /Users/dharamvir/Downloads/incubator-mxnet-master/tools/im2rec.py
/users/dharamvir/Desktop/training_rec /Users/dharamvir/Downloads/C-NMC_L
eukemia/preprocess_data/training_data --pass-through --pack-label --num-
thread 8

python /Users/dharamvir/Downloads/incubator-mxnet-master/tools/im2rec.py
/users/dharamvir/Desktop/validation_rec /Users/dharamvir/Downloads/C-NMC
_Leukemia/preprocess_data/validation_data  --recursive --list --num-thre
ad 8
python /Users/dharamvir/Downloads/incubator-mxnet-master/tools/im2rec.py
/users/dharamvir/Desktop/validation_rec /Users/dharamvir/Downloads/C-NMC
_Leukemia/preprocess_data/validation_data --pass-through --pack-label --
num-thread 8

aws s3 cp /Users/dharamvir/Desktop/recfiles/training.rec s3://leukemia-p
roject/training_data
aws s3 cp /Users/dharamvir/Desktop/recfiles/validation.rec s3://leukemia
-project/validation_data/
```

2. Create an Amazon SageMaker notebook instance

Once we have the data stores in the S3 buckets we need to set up a SageMaker notebook instance with a Spark cluster. This process will be done following the AWS documentation and it includes the following[15][16] :

a. Creating a notebook instance using Amazon EMR, we select a Spark cluster.
b. Store the data in S3 buckets
c. Load the image classification model. SageMaker is a platform based on Docker containers, thus every built-in algorithm is a Docker image already prepared with all the libraries that will be necessary. Since we are using transfer learning, which was explained in the documentation, we will use this algorithm as our model. However, we will have to use hyperparameter tuning to customize it to our data set.

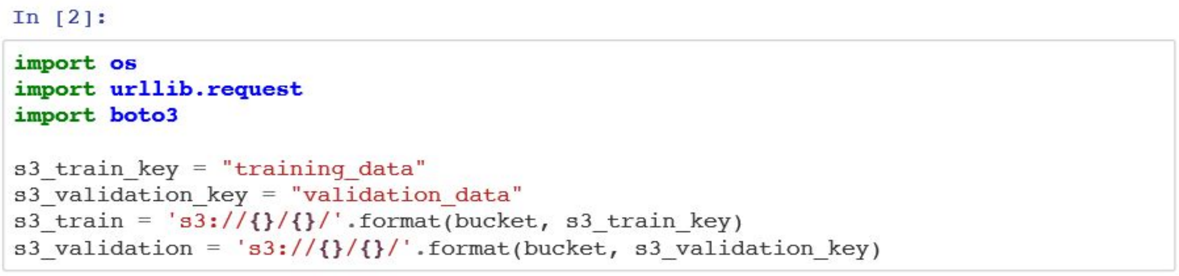


```
%%time
import boto3
import re
from sagemaker import get_execution_role
from sagemaker.amazon.amazon_estimator import get_image_uri

role = get_execution_role()
bucket='leukemia-project'
training_image = get_image_uri(boto3.Session().region_name, 'image-classification')
print(training_image)
```

```
825641698319.dkr.ecr.us-east-2.amazonaws.com/image-classification:1
CPU times: user 850 ms, sys: 188 ms, total: 1.04 s
Wall time: 6.44 s
```

3. **Pulling the data from the S3 buckets**

The images that were stored in the S3 bucket in RECORDIO format will be used for the training and validation of the model.



In [2]:

```
import os
import urllib.request
import boto3

s3_train_key = "training_data"
s3_validation_key = "validation_data"
s3_train = 's3://{}/{}/'.format(bucket, s3_train_key)
s3_validation = 's3://{}/{}/'.format(bucket, s3_validation_key)
```

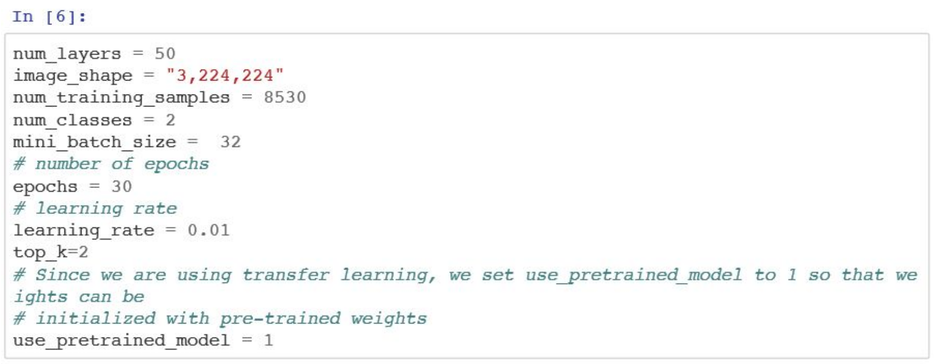
4. **Training the parameters of the algorithm**

Since we are using transfer learning, we will need to customize the parameters of the model to fit our data. The hyperparameters that are specific to the image classification algorithm are:

- **Num_layers:** The number of layers (depth) for the network. We will use 50.
- **Num_training_samples:** This is the number of training samples. In our case, once we separated the data in the training and testing, we were left with 8530 samples in the training set.
- **Num_classes:** This is the total number of classes. In our case, two, cancer and healthy cells.
- **Epochs:** Number of training epochs.
- **Learning_rate:** The learning rate at which the algorithm gets to a solution. In our case we decided a learning rate of 0.01.
- **Mini_batch_size:** The number of training samples used for each mini batch. Since we are using distributed training, the number of samples per batch will be N*mini_batch_size where N is the number of hosts on which the training is running.

Different values for these parameters were tried and we ended up choosing the ones stated above in order to have a better accuracy at the same that we didn't increase the computation time excessively.

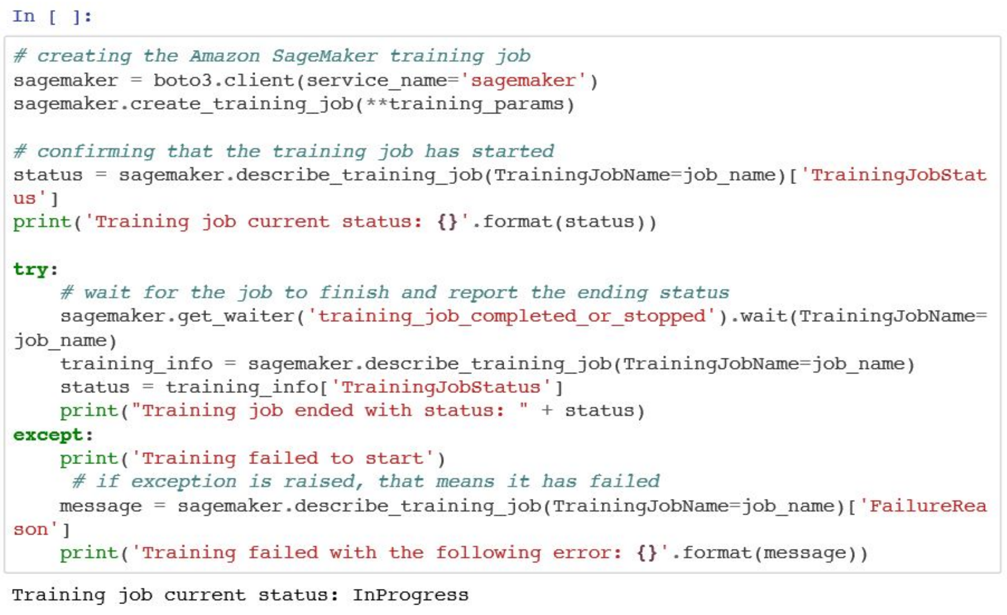
After setting the training parameters we kick off the training and poll for status until the model training is completed. For example, the training can take between 10 to 12 minutes per each epoch on a p2.xlarge machine. The network should typically converge after 10 epochs.



```
In [6]:
num_layers = 50
image_shape = "3,224,224"
num_training_samples = 8530
num_classes = 2
mini_batch_size =  32
# number of epochs
epochs = 30
# learning rate
learning_rate = 0.01
top_k=2
# Since we are using transfer learning, we set use_pretrained_model to 1 so that we
ights can be
# initialized with pre-trained weights
use_pretrained_model = 1
```

5. Training of the model

Once the model parameters have been stated, we will train the model using Amazon SageMaker CreateTrainingJob API. In each epoch the model will randomly select portions of the dataset for each batch, instead of using the whole dataset in each epoch. This will improve the final performance of the model, though it will introduce a small variation in the training results. We will create a job that will train our model and then it will be published in production as an endpoint.



```
In [ ]:
# creating the Amazon SageMaker training job
sagemaker = boto3.client(service_name='sagemaker')
sagemaker.create_training_job(**training_params)

# confirming that the training job has started
status = sagemaker.describe_training_job(TrainingJobName=job_name)['TrainingJobStat
us']
print('Training job current status: {}'.format(status))

try:
    # wait for the job to finish and report the ending status
    sagemaker.get_waiter('training_job_completed_or_stopped').wait(TrainingJobName=
job_name)
    training_info = sagemaker.describe_training_job(TrainingJobName=job_name)
    status = training_info['TrainingJobStatus']
    print("Training job ended with status: " + status)
except:
    print('Training failed to start')
     # if exception is raised, that means it has failed
    message = sagemaker.describe_training_job(TrainingJobName=job_name)['FailureRea
son']
    print('Training failed with the following error: {}'.format(message))

Training job current status: InProgress
```

With the Amazon CloudWatch Logs we will see all the training algorithm outputs. We will be able to see the information for the 'validation-accuracy'. With this value we will be able to see if the model is suffering from overfitting.

6. Deploying the model

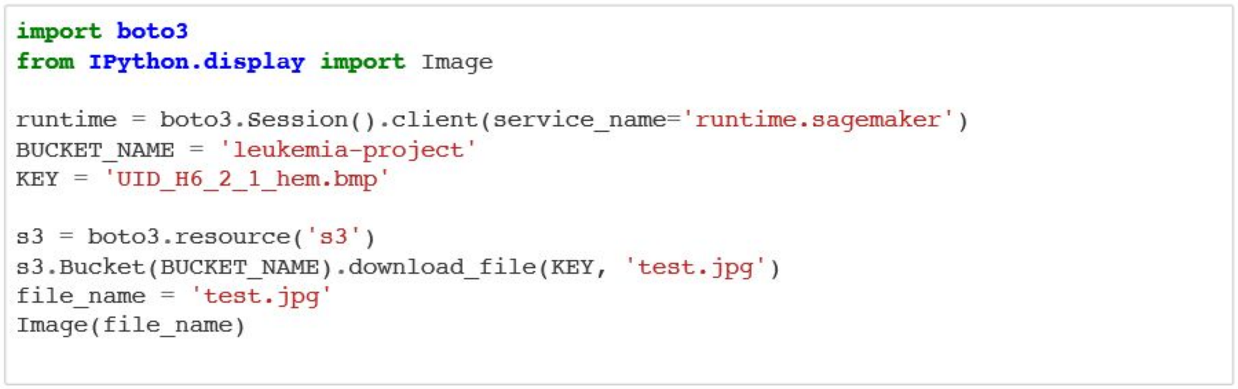
In order to deploy the model we will need to take the following steps:

a. Host the model: we will create a SageMaker model from the training output.
b. Create an endpoint configuration: A SageMaker Endpoint is a fully managed service that will allow us to make real-time inferences via a REST API.
c. Deploy the model using the endpoint configuration defined. SageMaker will create an instance and deploy a docker container with the algorithm that we selected. Inside this docker container the model will be hosted. The instance that we will create is a m4.xlarge, as it is the one that is provided by AWS. In our case, we are just using one model for our endpoint but many more models can be used with the same endpoint.

   The code for this part will be included in the appendix of the project. An endpoint is also monitored by CloudWatch so we can see information like the runtime logs, invocation metrics and instance metrics.

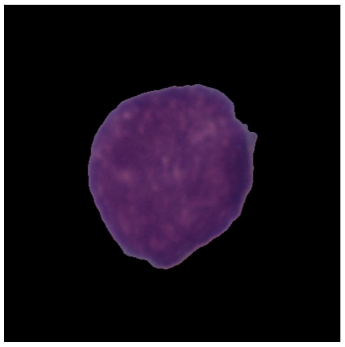
7. Perform real time inference

Once we have our model hosted, we will use it to make a prediction on a single image.



```python
import boto3
from IPython.display import Image

runtime = boto3.Session().client(service_name='runtime.sagemaker')
BUCKET_NAME = 'leukemia-project'
KEY = 'UID_H6_2_1_hem.bmp'

s3 = boto3.resource('s3')
s3.Bucket(BUCKET_NAME).download_file(KEY, 'test.jpg')
file_name = 'test.jpg'
Image(file_name)
```

Out[52]:



The model will be hosted for inference. The output of this will be the probability values for the classes encoded in JSON format. We will need to convert the JSON format into an array. Then, we will find the class with the maximum probability and we will print the class index.

In [53]:

```
import json
import numpy as np
with open(file_name, 'rb') as f:
    payload = f.read()
    payload = bytearray(payload)

response = runtime.invoke_endpoint(EndpointName=endpoint_name,
                                   ContentType='application/x-image',
                                   Body=payload)
result = response['Body'].read()
# result will be in json format and convert it to ndarray
result = json.loads(result)
print(result)
# the result will output the probabilities for all classes
# find the class with maximum probability and print the class index
index = np.argmax(result)
object_categories = ['all', 'hem']
print("Result: label - " + object_categories[index] + ", probability - " + str(resu
lt[index]))
```

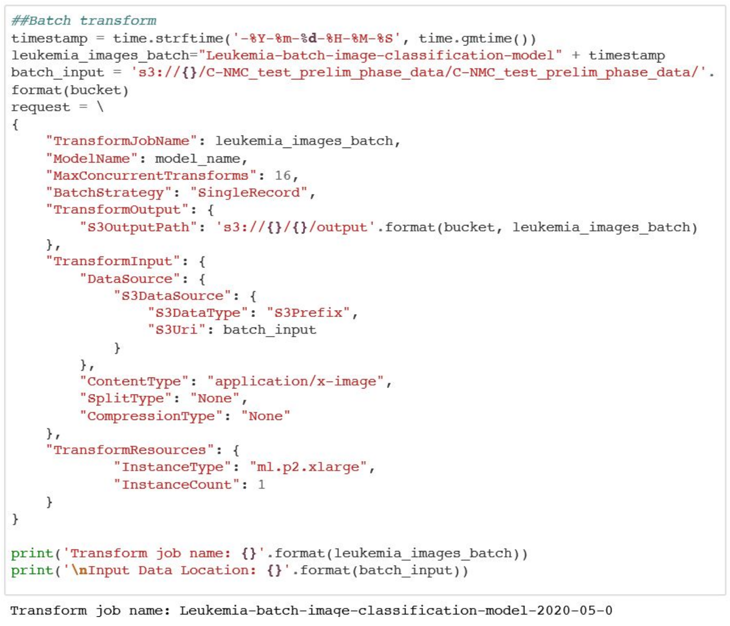
```
[0.5013220310211182, 0.49867793917655945]
Result: label - all, probability - 0.5013220310211182
```

8. Create a batch transform job

Batch transform will be used to perform a prediction in a big dataset or schedule. With all of our data in the S3 bucket, we will connect it to SageMaker which will start the batch transform job starting a new EC2 instance. This new instance will process all the data and return it back into the S3 bucket in addition to the inference done.

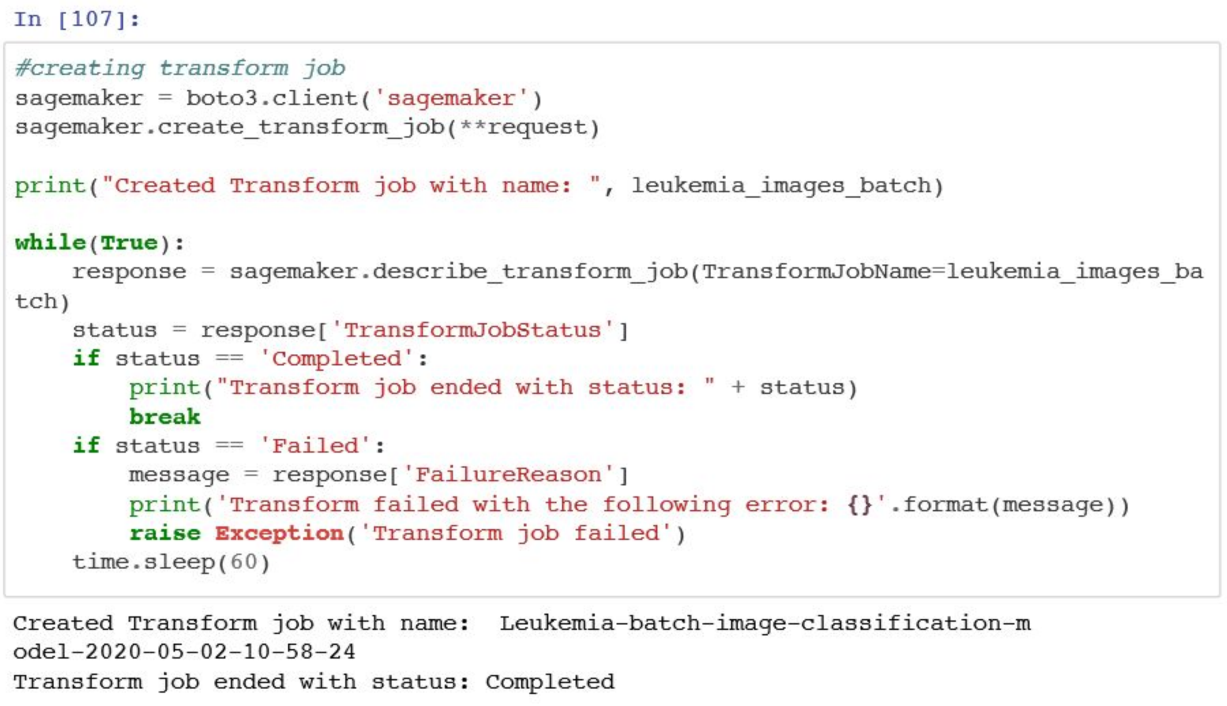
We created a batch establishing the parameters like the input and output location and the instance type, in our case p2.xlarge.

We create the transform job and we will be ready to perform the classification model on the images.



```
##Batch transform
timestamp = time.strftime('-%Y-%m-%d-%H-%M-%S', time.gmtime())
leukemia_images_batch="Leukemia-batch-image-classification-model" + timestamp
batch_input = 's3://{}/C-NMC_test_prelim_phase_data/C-NMC_test_prelim_phase_data/'.
format(bucket)
request = \
{
    "TransformJobName": leukemia_images_batch,
    "ModelName": model_name,
    "MaxConcurrentTransforms": 16,
    "BatchStrategy": "SingleRecord",
    "TransformOutput": {
        "S3OutputPath": 's3://{}/{}/output'.format(bucket, leukemia_images_batch)
    },
    "TransformInput": {
        "DataSource": {
            "S3DataSource": {
                "S3DataType": "S3Prefix",
                "S3Uri": batch_input
            }
        },
        "ContentType": "application/x-image",
        "SplitType": "None",
        "CompressionType": "None"
    },
    "TransformResources": {
            "InstanceType": "ml.p2.xlarge",
            "InstanceCount": 1
    }
}

print('Transform job name: {}'.format(leukemia_images_batch))
print('\nInput Data Location: {}'.format(batch_input))
```

```
Transform job name: Leukemia-batch-image-classification-model-2020-05-0
```

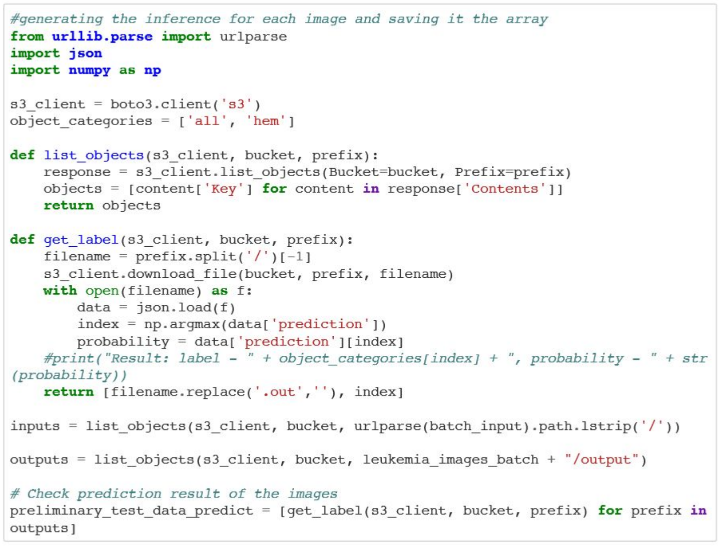


```
In [107]:

#creating transform job
sagemaker = boto3.client('sagemaker')
sagemaker.create_transform_job(**request)

print("Created Transform job with name: ", leukemia_images_batch)

while(True):
    response = sagemaker.describe_transform_job(TransformJobName=leukemia_images_ba
tch)
    status = response['TransformJobStatus']
    if status == 'Completed':
        print("Transform job ended with status: " + status)
        break
    if status == 'Failed':
        message = response['FailureReason']
        print('Transform failed with the following error: {}'.format(message))
        raise Exception('Transform job failed')
    time.sleep(60)
```

```
Created Transform job with name:  Leukemia-batch-image-classification-m
odel-2020-05-02-10-58-24
Transform job ended with status: Completed
```

9. Apply classification model to test data and calculate F1 score

We perform the prediction for our data, having a label of classification for each image. This will be stored in a data frame with the real label and the predicted label in order to obtain the F1 score. Since SageMaker does not provide this metric, we will have to calculate it with the confusion matrix. These results will be explained in more detail below in the results.



```
#generating the inference for each image and saving it the array
from urllib.parse import urlparse
import json
import numpy as np

s3_client = boto3.client('s3')
object_categories = ['all', 'hem']

def list_objects(s3_client, bucket, prefix):
    response = s3_client.list_objects(Bucket=bucket, Prefix=prefix)
    objects = [content['Key'] for content in response['Contents']]
    return objects

def get_label(s3_client, bucket, prefix):
    filename = prefix.split('/')[-1]
    s3_client.download_file(bucket, prefix, filename)
    with open(filename) as f:
        data = json.load(f)
        index = np.argmax(data['prediction'])
        probability = data['prediction'][index]
    #print("Result: label - " + object_categories[index] + ", probability - " + str
(probability))
    return [filename.replace('.out',''), index]

inputs = list_objects(s3_client, bucket, urlparse(batch_input).path.lstrip('/'))

outputs = list_objects(s3_client, bucket, leukemia_images_batch + "/output")

# Check prediction result of the images
preliminary_test_data_predict = [get_label(s3_client, bucket, prefix) for prefix in
outputs]
```

B. Experiment using a deep learning framework

For this second experiment, we decided to use one of the existing deep learning frameworks. Upon reading documentation about the different possibilities available we decided to try and use the Databricks library SparkDL.

1. Setting up EMR notebook and installing libraries:

SparkDL like it was mentioned in the documentation section is a deep learning library that uses Keras and Tensorflow as its backend. It is also a library that uses transfer learning with pre-trained models with Spark and is available in both batch and streaming data processing. This library allows us to run single node models in a distributed fashion on large amounts of data.

After creating an EMR notebook with a Spark instance, we needed to get the libraries installed in it. We used the new feature of EMR notebooks that allows us to install on a running cluster. The installation happens on the notebook file, so that if we need to change clusters the library environment can be recreated. We were able to install both Keras and Tensorflow starting the kernel to PySpark as these libraries were part of the PyPi repository.

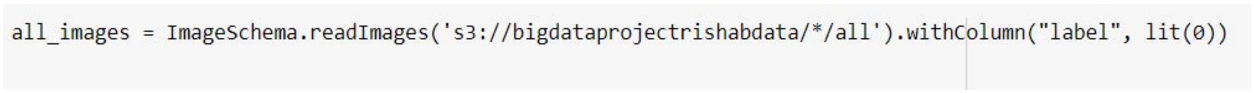
However, when trying to get SparkDL installed into the notebook we runned into some issues. The SparkDL library version that can be installed into an EMR notebook is an outdated version that doesn't have all the modules necessary to work. Therefore, it was not possible for us to run in on a Spark EMR notebook. Also, many of the functions of the package are obsolete and no longer work. Having this problem that had no apparent solution, we decided to change the deep learning framework we would be using and decided to create our model using Tensorflow on Spark.

2. Change of deep learning framework from SparkDL to Tensorflow

In the previous step we installed all the libraries necessary using *sc.install_pypi_package()* command including both Tensorflow and Keras. We will be using transfer learning with a pre-trained model of the Tensorflow library called ResNet50. This model is a CNN that has been previously trained for another image dataset. With a featurizer function we will peel off the last layer of the pre-trained CNN from ResNet50 and we will use the outputs of the previous layers as the features for a logistic regression algorithm. This logistic regression comes from the Spark package MLlib, which contains many machine learning algorithms to be used in Spark. With this method, the algorithm can converge using fewer images than building the model from the beginning.

3. Creating the Training and Test Sets

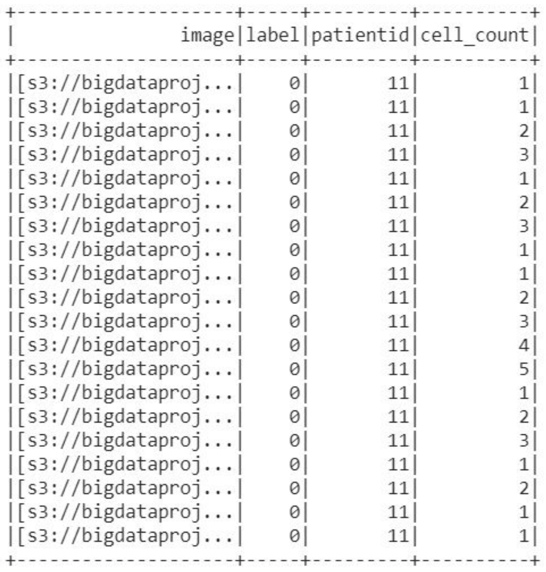
Like in the previous experiment, we have all of our images inside an AWS S3 bucket. We need to pull these images from the bucket to create our training and testing sets. We used the MLlib function *ImageSchema.readImages* which allowed us to read the images from the S3 bucket into a Pyspark SQL dataframe.



```
all_images = ImageSchema.readImages('s3://bigdataprojectrishabdata/*/all').withColumn("label", lit(0))
```

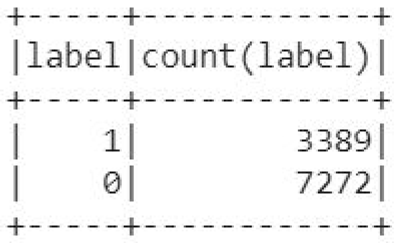
We read the images into the schema and we add a label to show if the image belongs to a healthy category or if it belongs to the cancer category.

Now that we have all the images of both cancer and healthy cells in a data frame we need to merge them together so that we can separate the images by patient ID. This way as we did in the SageMaker model, we will have a training set containing a percentage of the patients and a testing containing the rest of the patients.

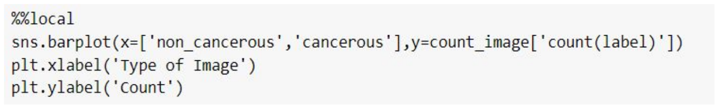


```
+--------------------+-----+---------+----------+
|               image|label|patientid|cell_count|
+--------------------+-----+---------+----------+
|[s3://bigdataproj...|    0|       11|         1|
|[s3://bigdataproj...|    0|       11|         1|
|[s3://bigdataproj...|    0|       11|         2|
|[s3://bigdataproj...|    0|       11|         3|
|[s3://bigdataproj...|    0|       11|         1|
|[s3://bigdataproj...|    0|       11|         2|
|[s3://bigdataproj...|    0|       11|         3|
|[s3://bigdataproj...|    0|       11|         1|
|[s3://bigdataproj...|    0|       11|         1|
|[s3://bigdataproj...|    0|       11|         2|
|[s3://bigdataproj...|    0|       11|         3|
|[s3://bigdataproj...|    0|       11|         4|
|[s3://bigdataproj...|    0|       11|         5|
|[s3://bigdataproj...|    0|       11|         1|
|[s3://bigdataproj...|    0|       11|         2|
|[s3://bigdataproj...|    0|       11|         3|
|[s3://bigdataproj...|    0|       11|         1|
|[s3://bigdataproj...|    0|       11|         2|
|[s3://bigdataproj...|    0|       11|         1|
|[s3://bigdataproj...|    0|       11|         1|
+--------------------+-----+---------+----------+
```

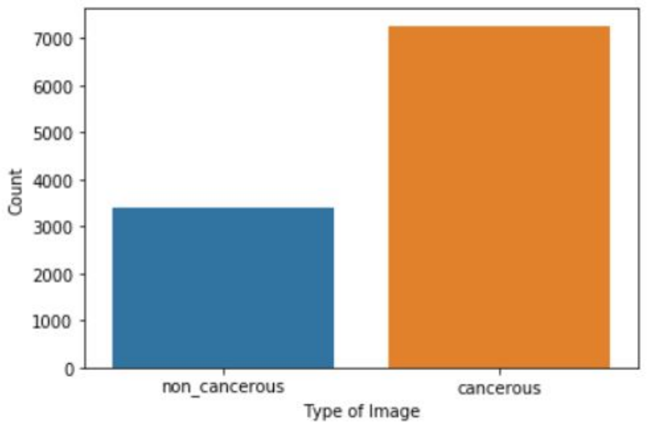
Once we have all the images with their classifying label, the patient ID and the number of cells in each image, we can do a bit of exploratory analysis to see how many images we have of each class.



```
+-----+------------+
|label|count(label)|
+-----+------------+
|    1|        3389|
|    0|        7272|
+-----+------------+
```



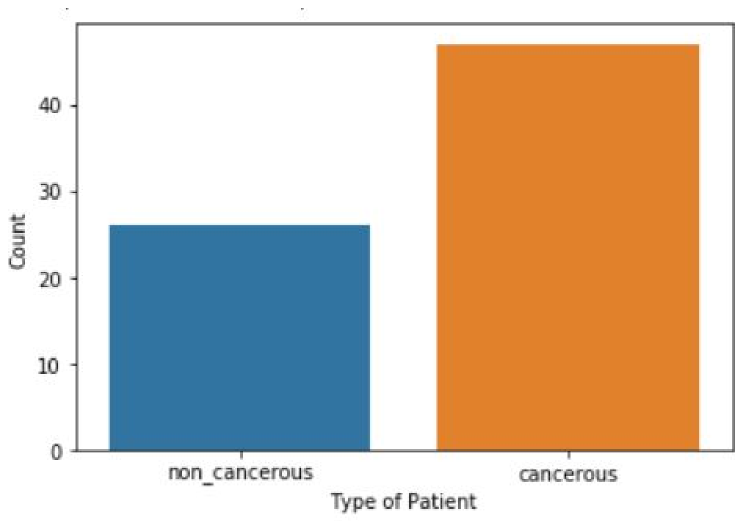
```
%%local
sns.barplot(x=['non_cancerous','cancerous'],y=count_image['count(label)'])
plt.xlabel('Type of Image')
plt.ylabel('Count')
```



We can see that we have 3389 images of healthy cells and 7272 images of cancer cells in our data set.

We can also count instead of the number of images that are healthy or malignant, we can see the number of patients that are healthy and those who are sick.

```
%%local
sns.barplot(x=['non_cancerous','cancerous'],y=count_patient['count(DISTINCT patientid)'])
plt.xlabel('Type of Patient')
plt.ylabel('Count')
```

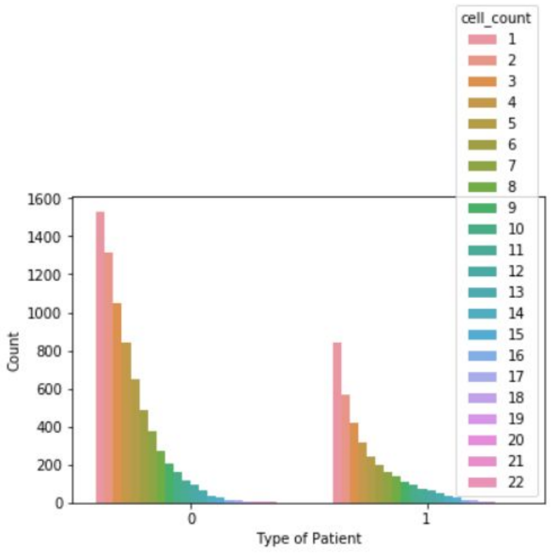


Finally, we can count the number of cells that the dataset contains depending if the patient is healthy or sick.

```
%%local
sns.barplot(x=count_cell['label'],y=count_cell['count(label)'],hue = count_cell['cell_count'])
plt.xlabel('Type of Patient')
plt.ylabel('Count')
```

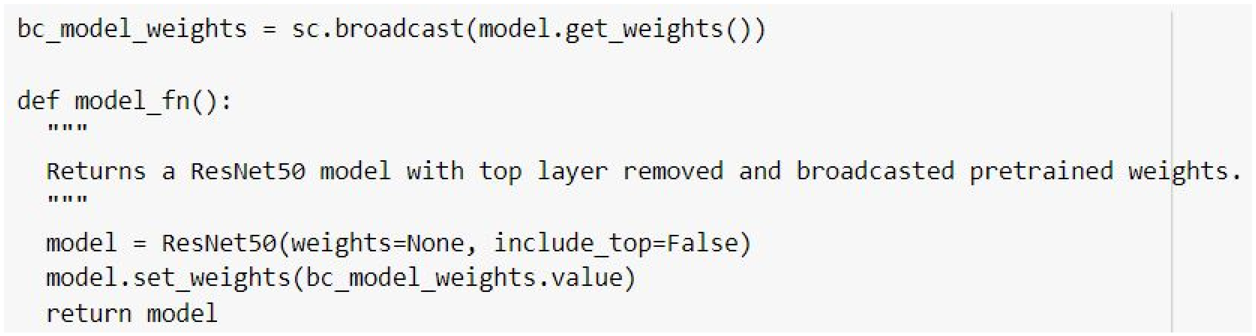
We can see how we have a larger cell count if the patient has cancer (label 0) than if the patient is healthy.

We will split our patients in 80-20% for the training and test sets.



4. Creating the model with ResNet50

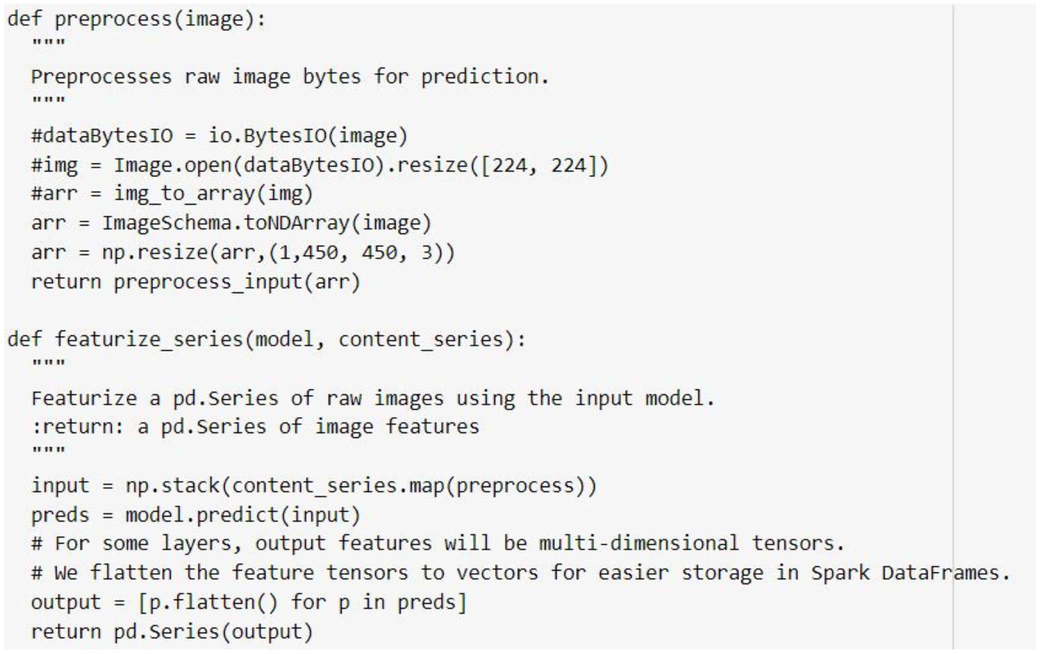
Once we have our training and testing sets, we will create our model for the transfer learning. Like it was mentioned before we will be using ResNet50 as our pre-trained model and then we will use the output of this pre-trained model as the features for a logistics regression.



```
bc_model_weights = sc.broadcast(model.get_weights())

def model_fn():
  """
  Returns a ResNet50 model with top layer removed and broadcasted pretrained weights.
  """
  model = ResNet50(weights=None, include_top=False)
  model.set_weights(bc_model_weights.value)
  return model
```

5. Creating the features with a featurizer function

Once we have created our ReNet50 model we will need to preprocess our images so that they have the correct format. For this we will create a preprocess function that we will use for our images. We also create a featurizing function that will take the output of the ResNet50 model and return them as the features we will use for the logistic regression.



```
def preprocess(image):
  """
  Preprocesses raw image bytes for prediction.
  """
  #dataBytesIO = io.BytesIO(image)
  #img = Image.open(dataBytesIO).resize([224, 224])
  #arr = img_to_array(img)
  arr = ImageSchema.toNDArray(image)
  arr = np.resize(arr,(1,450, 450, 3))
  return preprocess_input(arr)

def featurize_series(model, content_series):
  """
  Featurize a pd.Series of raw images using the input model.
  :return: a pd.Series of image features
  """
  input = np.stack(content_series.map(preprocess))
  preds = model.predict(input)
  # For some layers, output features will be multi-dimensional tensors.
  # We flatten the feature tensors to vectors for easier storage in Spark DataFrames.
  output = [p.flatten() for p in preds]
  return pd.Series(output)
```

```
@pandas_udf(ArrayType(DoubleType(), containsNull=False), PandasUDFType.SCALAR_ITER)
def featurize_udf(content_series_iter):
  '''
  This method is a Scalar Iterator pandas UDF wrapping our featurization function.
  The decorator specifies that this returns a Spark DataFrame column of type ArrayType(FloatType).

  :param content_series_iter: This argument is an iterator over batches of data, where each batch
                              is a pandas Series of image data.
  '''
  # With Scalar Iterator pandas UDFs, we can load the model once and then re-use it
  # for multiple data batches.  This amortizes the overhead of loading big models.
  model = model_fn()
  for content_series in content_series_iter:
    yield featurize_series(model, content_series)
```

We create another featurize function using Pandas Scalar Iterator that will work iteratively so that we can work with batches of images. This will help us with the reducing training time as the model will not have to re-initialize every time the featurizing function is called.

Once we have defined the functions, we will create the features for the training set.

```
features_train_df = train.repartition(16).select(col("label"), featurize_udf("image").alias("features"))
```

6. Creating the logistic regression model

With these features we will use them in the fitting of the logistic regression model.

```
features_train_df = features_train_df.select(
    features_train_df["label"],
    list_to_vector_udf(features_train_df["features"]).alias("features")
)
```

```
lr = LogisticRegression(maxIter=5, regParam=0.03,
                        elasticNetParam=0.5,labelCol="label")
```

```
logisticRegressionModel = lr.fit(features_train_df)
```

7. Test model on testing set

With our logistic regression fit we can make predictions. We will see how the model behaves with new data from the testing set and we will calculate the F1 score of this model.

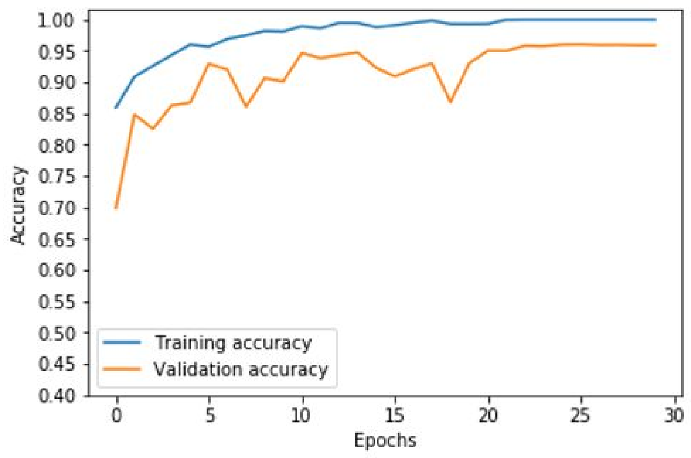
Results and Conclusions

Results

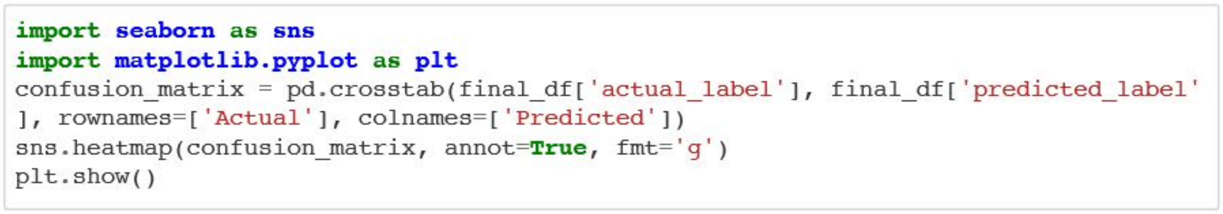
In the section above the two experiments that were conducted were explained thoroughly. In this section, we will summarize the main results obtained, the problems encountered and we will extract some conclusions from the experiments of the project.

A. SageMaker Experiment:

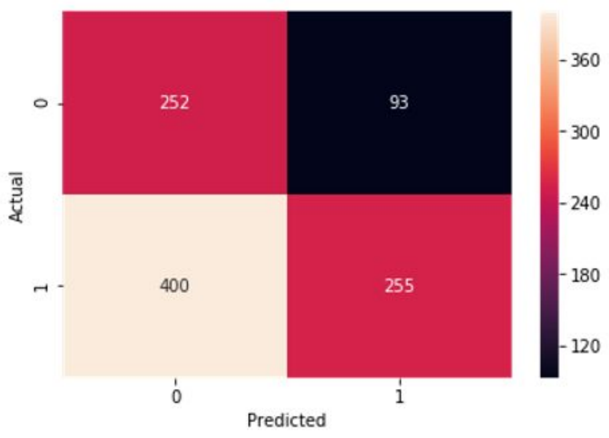
The SageMaker model had a run time of approximately two hours. In the training and validation of the model we got the following results:



We can see how our model had a very high accuracy for both the training and the validation sets. Taking into consideration that our dataset was unbalanced we confirm that using the accuracy metric is not the best metric to see our models performance. That is why for our testing we calculate the confusion matrix to get the F1. We did not do this for the training since SageMaker does not return the F1 score of the model. Since for the testing the model was deployed we were able to calculate the F1 score of the testing.



```
import seaborn as sns
import matplotlib.pyplot as plt
confusion_matrix = pd.crosstab(final_df['actual_label'], final_df['predicted_label'
], rownames=['Actual'], colnames=['Predicted'])
sns.heatmap(confusion_matrix, annot=True, fmt='g')
plt.show()
```



Taking into consideration the results of the confusion matrix we calculated the accuracy and F1 obtaining:

- **Accuracy:** 0.507
- **F1 Score:** 0.5084

We can see that this metric is much lower than that obtained in the training set. This could be for a number of various reasons, from overfitting of the model to the training set and the unbalanced data. However, we were able to run a deep learning model in a Spark instance which was the objective of the project.

B. Tensorflow Experiment:

The Tensorflow experiment ran into some major problems from the get go. Firstly we had some problems when installing the deep learning framework that we planned on using, SparkDL. After changing to using Tensorflow we were able to get some work done as explained above. However, we ran into another major problem.

In the methodology we described how we were going to create the features to input into the Logistic Regression model. However, we had a problem with this as the Pandas Scalar Iterator. This function has been depreciated, thus we were unable to featurize our images to create our inputs for the Logistics Regression model. Running the process without an iterator led to excessive ram consumption as the model had to be initialized at every call of the function which led to high training times and high ram consumption. Hence we decided to move away from the tensorflow implementation to focus on our Sage-maker implementation.

Project Outtakes

With this project we tried to explore the uses of Spark in a deep learning framework. In the beginning we planned on using an existing framework that ran on Spark to create a classification model that would tell us if a patient had cancer or was healthy. Working in a distributed system with Deep Learning is a field which still has a lot of future work to be done as there are many functions that have problems working. We had a lot of problems trying to get various of the deep learning frameworks we found in the documentation to work properly. But we did learn that there are many options to work with deep learning in order to process images even if we were unable to implement the model as we would have wished.

On the other hand, we used a new technology to us, Amazon SageMaker. With this project we had to learn how to use this new technology. We ran into some complications but in the end we were able to create a model and deploy it. The results we obtained were not extremely pleasing but that is due to the preparation of the data as we had an unbalanced data set. Nevertheless, it allowed us to explore a new technology and learn from it.

References

[1]Andreas Mainer, Christopher Syben, Tobias Lasser, Christian Riess, "*A gentle introduction to deep learning in medical image processing*", Special Issue: Deep Learning in Medical Physics. May 2009

[2] Bernard Marr "*How much data do we generate every day? The mind-blowing stats everyone should read*", Forbes Magazine

[3] Matei Zaharia, Reynold S. Xin, Patrick Wndell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen Shivaram Venkataramn, Michael J Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker and Ion Stoica, "*Apache Spark: A Unified Engine for Big Data Processing*". Communications of the ACM Vol.59 Nº11, November 2016

[4]Matei Zaharia, Reynold S. Xin, Patrick Wndell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen Shivaram Venkataramn, Michael J Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker and Ion Stoica, "*Apache Spark: A Unified Engine for Big Data Processing*". Communications of the ACM Vol.59 Nº11, November 2016

[5]yahoo/TensorFlowOnSpark: TensorFlowOnSpark brings TensorFlow programs to Apache Spark clusters.

[6] yahoo/CaffeOnSpark: Distributed deep learning on Hadoop and Spark clusters.

[7]maxpumperla/elephas: Distributed Deep learning with Keras & Spark

[8]https://github.com/intel-analytics/BigDL

[9]databricks/spark-deep-learning: Deep Learning Pipelines for Apache Spark

[10] Sue Ann Hong, Tim Hunter and Reynold Xin, "*A Vision for making deep learning simple. From Machine learning practitioners to business analysts*" Engineering Blog, June 6 2017

[11] Dipanjan Sarkar, "*A comprehensive Hands-on Guide to Transfer Learning with Real-World Applications in Deep Learning*", Towardsdatascience.com, Nov 14 2018

[12]Run Jupyter Notebook and JupyterHub on Amazon EMR

[13] *"Amazon SageMaker for Machine Learning Deep Learning- a comprehensive guide"* Medium. December 12 2019

[14]https://competitions.codalab.org/competitions/20395#learn_the_details-data-description

[15]https://docs.aws.amazon.com/sagemaker/latest/dg/nbi.html

[16]https://aws.amazon.com/es/blogs/machine-learning/classify-your-own-images-using-amazon-sagemaker/

Appendix: Code

[17] big_data.pdf

[18] bigdataprojectviusalization.ipynb