UNIVERSITÀ DELLA CALABRIA

DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE

Deep learning Project

MEDICAL IMAGING RSNA SCREENING MAMMOGRAPHY BREAST CANCER DETECTION & CONTRADICTORY TEXT ANALYSIS

22 June 2023

Authors:
Muhammad Waseem Akram
Muhammad Fiaz
Mudassar Ahmad

Academic Year 2022/2023

TABLE OF CONTENTS

Chapter 1

Medical Imaging RSNA Screening Mammography Breast Cancer Detection

In this chapter, we will discuss about the Breast cancer detection model construction by using RSNA screening Mammographs dataset, we take the data set of mammograph which consists of train and test mammograph in the form of dicom images samples and train csv file that have detail information of train and test mammograph. we will discuss in detail the preprocessing of mammographs for model, design of deep learning model and it's training on the RSNA cancer dataset. The deep learning model and techniques can be helpful to detect the breast cancer using the mammographs.

1.1 Introduction

Cancer is the second leading cause of death globally, accounting for about one in every six deaths reported worldwide. Breast cancer (BC) is the second most reported cancer, with about 2.09 million reported cases and 627,000 deaths in 2018 alone. Although the incidence rate is reducing in developed countries, the reverse is the case in low- and middle-income countries; for instance, African countries accounted for 50 % of the reported cases and 58% of deaths in 2018. Moreover, BC survival rate has increased to about 80 % in North America (70% among black women on the continent) and 60% in Sweden and Japan, whereas it remains less than 40% in low-income countries. This is because the low- and middle-income countries have inadequate health management facilities such as diagnosis and treatment facilities; this results in late detection and late-stage treatment among women with the disease.

Breast cancer is common among women, although a few cases among men have been reported. BC is a malignant growth that starts from either the lobules or the milk duct of the breast. Ductal carcinoma in situ (DCIS) is a precancerous condition that begins its growth and is contained in the milk duct; it is considered the earliest appearance of BC and is easily detected by breast exam. Similarly, lobular carcinoma in Stu (LCIS) is an abnormal growth that begins and is contained in the milk-producing lobule cells but does not invade or spread to other parts of the breast. However, unlike DCIS, LCIS is not easily detected by breast exam. BC can be invasive; a cancer that begins in the milk duct but spreads to other parts of the breast is called invasive ductal carcinoma (IDC), while one that grows from the lobule cells and then spreads to the other parts of the breast is called invasive lobular carcinoma (ILC). Lastly, BC can be metastatic when the cancer cells penetrate the circulatory or lymph system, spreading to other parts of the body via the bloodstream.

Early diagnosis of BC has been found to constrain cancer growth, prevent spreading, ease treatment, and reduce the mortality rate by 25%. BC diagnosis techniques include breast exams, biopsy, mammograms, breast ultrasound, and magnetic resonance imaging (MRI). Digital mammographic screening is the most common, cheapest, and most effective BC screening technology capable of detecting up to 90% BC even before a lump can be felt by breast exam. It uses a low-dose X-ray imaging of the breast where tissues in the breast, including tumors, appear as different shades of gray on the image. This makes mammogram screening the choice diagnostic technique in low-

and middle-income countries.

In diagnosing BC from a mammogram, radiologists look for specific abnormalities such as architectural distortion of breast tissue, alignment of the two breasts, masses, and calcification. Mammograms of the two breasts are taken from two views—the craniocaudal view (top-bottom view) and mediolateral oblique (MLO) view—to give the radiologist a comprehensive view for the examination . Radiologists interpret their diagnosis using a standardized breast imaging reporting and data system (BI-RADS) scale developed by the American College of Radiology (ACR). The BI-RADS scale ranges from categories 0 to 6 detailed in Table

However, due to low contrast, mammogram images are among the most difficult medical images to analyze. The sensitivity of mammograms is greatly affected by breast density and fats, which are radiolucent; hence, their appearance is similar to mass or calcification in the image. As a result, the sensitivity of mammograms to early detection and accurate diagnosis has been estimated at 85–90%. Today, medical centers face the challenge of screening an increasingly high volume of mammograms for accurate diagnosis, including early detection. To assist the radiologist, computer-aided diagnostic (CAD) systems have been proposed to reduce misdiagnosis. The developed CAD systems are based on different machine learning techniques . However, the most successful of these techniques are based on deep convolution neural networks (CNNs) for the detection of mammograms [12,13]. While this method has produced very commendable results, it suffers from data availability. Deep CNN models require substantial training data (in order of hundreds of millions) to achieve high accuracy, sensitivity, and specificity; meanwhile, the medical image dataset is usually scanty (available in tens of thousands). In addition to data availability, deep CNN requires high computational power. This twin problem has greatly limited the clinical application of these models.

Currently, early detection of breast cancer requires the expertise of highly-trained human observers, making screening mammography programs expensive to conduct. A looming shortage of radiologists in several countries will likely worsen this problem. Mammography screening also leads to a high incidence of false positive results. This can result in unnecessary anxiety, inconvenient follow-up care, extra imaging tests, and sometimes a need for tissue sampling (often a needle biopsy).

Our efforts in this project could help extend the benefits of early detection to a broader popula-

tion. Greater access could further reduce breast cancer mortality worldwide.

1.2 Objective of project

1. The goal of this project is to build deep learning model to identify breast cancer. We will train our model with screening mammograms obtained from regular screening and classify into to cancer = 1 and non-caner = 0 groups.

2. Our work improving the automation of detection in screening mammography may enable radiologists to be more accurate and efficient, improving the quality and safety of patient care. It could also help reduce costs and unnecessary medical procedures.

1.3 Dataset Description

We are using the RSNA Screening Mammography Breast Cancer Detection dataset of ongoing RSNA competition on kaggle. The data set consists of dicom images mammography of patients and the csv file that have the details of patients mammography and related labels. We upload the breast cancer PNG images from Kaggel that are already convert from breast cancer dicom . We call the dataset by following code

```
path_csv = '/kaggle/input/rsna-breast-cancer-detection'
path_image='/kaggle/input/rsna-breast-cancer-512-pngs'

# load the csv_file
train_df =pd.read_csv(os.path.join(path_csv, 'train.csv'))
test_df =pd.read_csv(os.path.join(path_csv,'test.csv'))

#define the images path
train_dir = os.path.join(path_csv,'train_images')
test_dir = os.path.join(path_csv,'test_images')
display(train_df)
```

	site_id	patient_id	image_id	laterality	view	age	cancer	biopsy	invasive	BIRADS	implant	density	machine_id	difficult_negative_case
0	2	10006	462822612	L	CC	61.0	0	0	0	NaN	0	NaN	29	False
1	2	10006	1459541791	L	MLO	61.0	0	0	0	NaN	0	NaN	29	False
2	2	10006	1864590858	R	MLO	61.0	0	0	0	NaN	0	NaN	29	False
3	2	10006	1874946579	R	CC	61.0	0	0	0	NaN	0	NaN	29	False
4	2	10011	220375232	L	CC	55.0	0	0	0	0.0	0	NaN	21	True
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
54701	1	9973	1729524723	R	MLO	43.0	0	0	0	1.0	0	C	49	False
54702	1	9989	63473691	L	MLO	60.0	0	0	0	NaN	0	C	216	False
54703	1	9989	1078943060	L	CC	60.0	0	0	0	NaN	0	C	216	False
54704	1	9989	398038886	R	MLO	60.0	0	0	0	0.0	0	C	216	True
54705	1	9989	439796429	R	CC	60.0	0	0	0	0.0	0	C	216	True

54706 rows × 14 columns

The dataset have 54,706 dicom images for training of 11,913 patients on the 10 differnet machine at two different location. The training set consist of 53,548 non cancer and 1,158 cancer mamomgraphs.

```
The train shape (54706, 14)
The number of unique images = 54706
The number of patients = 11913
The number of unique machine = 10
The number of unique view = 6
The number of unique sites = 2
The number of cancer images 1158
The number of Non-cancer images 53548
```

	site_id	patient_id	image_id	laterality	view	age	cancer	biopsy	invasive	BIRADS	implant	density	machine_id	difficult_negative_case
0	2	10006	462822612	L	CC	61.0	0	0	0	NaN	0	NaN	29	False
1	2	10006	1459541791	L	MLO	61.0	0	0	0	NaN	0	NaN	29	False
2	2	10006	1864590858	R	MLO	61.0	0	0	0	NaN	0	NaN	29	False
3	2	10006	1874946579	R	CC	61.0	0	0	0	NaN	0	NaN	29	False
4	2	10011	220375232	L	CC	55.0	0	0	0	0.0	0	NaN	21	True

1.4 Methodology

To build the deep learning model for cancer image detection. We will use the pre-train model ResNet50 V2 with necessary changes in the top layers. For the training, we are using the balance part of dataset which give the equal opportunity to model for learning the parameter. Before the training, we will preprocessing the images dataset in the meaningful representation that can accept our model as the input data. We will give the more details in the following subsections.

1.4.1 Data Set Preprocessing

We will use the balance subset of given dataset to train our model. To prepare balance dataset from the given dataset, we check the number of cancer images with respect to their laterality. After that we consider the same number of non-cancer images with respect to laterality categories.

```
train_cancer= train_df[train_df.cancer == 1]
display(train_cancer.shape)
train_cancer.laterality.value_counts()
train_non_cancer= train_df[train_df.cancer==0]
train_non_cancer_L=train_non_cancer[train_non_cancer.laterality=='L'][:588]
display(train_non_cancer_L.shape)
train_non_cancer_R=train_non_cancer[train_non_cancer.laterality=='R'][:570]
display(train_non_cancer_R.shape)
train_n_cancer=pd.concat([train_non_cancer_L,train_non_cancer_R])
train_n_cancer.reset_index(drop=True, inplace=True)
display(train_n_cancer.shape)

#output
```

```
train_cancer
(1158,14)

laterality
L    588
R    570
Name: count, dtype: int64
(588, 14)
(570, 14)
(1158,14)
```

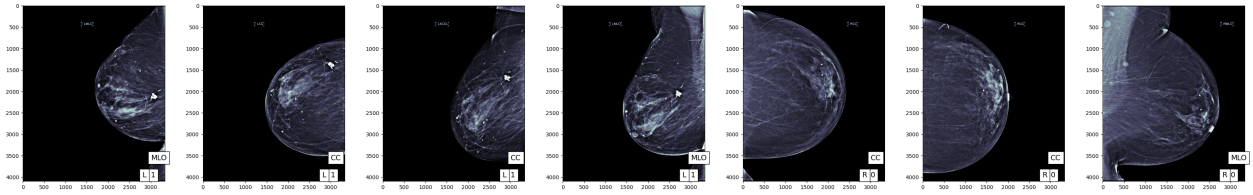
After making the train-n-cancer dataframe,we concatenate it with the train-cancer dataframe to make train-set that we will use to make dataset for training.

```
train_set= pd.concat([train_n_cancer_R,train_cancer])
train_set.reset_index(drop=True, inplace=True)
display(train_set.shape)

#output
(2316,14)
```

Our task is to take images and classify them into cancer or non-cancer classes. So, we select the images by use the image-id, patient-id, in train set. As we call the images of first patient who have cancer in the train-set shown below.

```
patient_id = train_df[train_df.cancer == 1].iloc[0].patient_id
one_patient_df = train_df[train_df.patient_id == patient_id]

images_dir = '/kaggle/input/rsna-breast-cancer-detection/{}_images/{}/{}.dcm'
n_rows = len(one_patient_df)
plt.figure(figsize=(6 * n_rows, 6))
for i in range(n_rows):
    row = one_patient_df.iloc[i]

    plt.subplot(1, n_rows, i + 1)

    img_arr = dicomsdl.open(images_dir.format('train', row.patient_id,
                    row.image_id)).pixelData()
    plt.imshow(img_arr, cmap = plt.cm.bone)
    plt.text(3000, 3600, row['view'], fontsize = 14,
                            bbox={'facecolor': 'white', 'pad' : 5})
    plt.text(3000, 4000, row['cancer'], fontsize = 14,
                            bbox={'facecolor': 'white', 'pad' : 5})
```

```
    plt.text(2800, 4000, row['laterality'], fontsize = 14,
                              bbox={'facecolor': 'white', 'pad' : 5})
```



Now, we select the images from the given dataset of images for training dataset and resize the images from 512×512 to 256×256 then save in the class folder 1 or 0 according to cancer status.

```
image_width = 256
image_height = 256

p_id = train_set.patient_id
i_id = train_set.image_id
cancer = train_set.cancer
for pid, iid, cncr in tqdm(zip(p_id, i_id, cancer)):
    tmpFile = str(pid) + "_" + str(iid) + ".png"
    tmpSrc = "/kaggle/input/rsna-breast-cancer-512-pngs/" + tmpFile
    tmpDst = "/kaggle/working/input_images/" + str(cncr) + "/" + tmpFile
    shutil.copyfile(tmpSrc, tmpDst)
    img = Image.open(tmpDst)
    img = img.resize((image_width, image_height), Image.ANTIALIAS)
    img.save(tmpDst)
```

After save the train-images in $'/kaggle/working/input-images/'$ directory we will preprocess the image from Gray scale to RGB because our model only take the tensor of the form $(None, 256, 256, 3)$ and split the train-ds into 80% train and 20 validation-ds. We train the model by using train-ds dataset and check the validation accuracy and loss on unseen dataset validation-ds during the training.

```
train_ds = tf.keras.preprocessing.image_dataset_from_directory(
    "/kaggle/working/input_images/",
    color_mode = "rgb",
    image_size = (256, 256),
    shuffle = True,
    validation_split = 0.25,
    subset = "training",
    seed = 2023)
```

```
validation_ds = tf.keras.preprocessing.image_dataset_from_directory(
    "/kaggle/working/input_images/",
    color_mode = "rgb",
    image_size = (256, 256),
    shuffle = True,
    validation_split = 0.2,
    subset = "validation",
    seed = 70)


    #output

Found 2316 files belonging to 2 classes.
Using 1737 files for training.

Found 2316 files belonging to 2 classes.
Using 463 files for validation.

#train_ds
    <_BatchDataset element_spec=(TensorSpec(shape=(None, 256, 256, 3),
                dtype=tf.float32, name=None), TensorSpec(shape=(None,),
                dtype=tf.int32, name=None))>
```

1.4.2 Architecture and Training

We are using the pre trained model ResNet50V2 as the base model with already trained parameter.

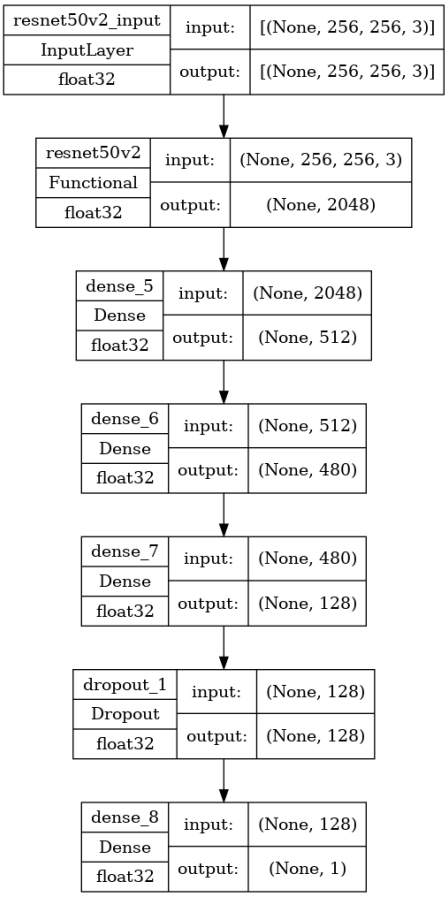
```
with strategy.scope():
    base_model = tf.keras.applications.resnet_v2.ResNet50V2(
    include_top=False,
    weights="imagenet",
    input_shape=(image_height, image_width, 3),
    pooling="max")
    for layer in base_model.layers:
        layer.trainable = False
```

We build our model by using the pre trained model ResNet50V2 as the base of our model without his top layer and add the four dense layer with Relu activation function, one Dropout layer with 0.3 and last layer of one unit with Sigmoid activation function. We use the adam optimizer and binary crossentropy as a loss function in our model.

```
def model_tu(HP):
    with strategy.scope():
        model = tf.keras.Sequential()
        model.add(base_model)
        model.add(tf.keras.layers.Dense(512, activation = 'relu'))
         # Choose an optimal value between 32-512
        hp_units = HP.Int('units', min_value=32, max_value=512, step=32)
        model.add(keras.layers.Dense(units=hp_units, activation='relu'))
        model.add(tf.keras.layers.Dense(256, activation = 'relu'))
        model.add(tf.keras.layers.Dense(128, activation = 'relu'))
        model.add(tf.keras.layers.Dropout(0.3))
        model.add(tf.keras.layers.Dense(1, activation = 'sigmoid'))

        for i, layer in enumerate(model.layers):
            if(layer.name == "ResNet50V2"):
                layer.trainable = False

        hp_le_rate = HP.Choice('learning_rate', values=[1e-4, 1e-5, 1e-6, 1e-7])
        model.compile(optimizer=keras.optimizers.Adam(learning_rate=hp_le_rate),
                      loss='binary_crossentropy',
                      metrics=['accuracy'])
        return model
```

Using the Hyperband tuner from the Keras Tuner library (kt) to perform hyper parameter tuning on the model to optimize the hyper parameter for the best validation accuracy with 20 epochs to train the model for each hyperparameter configuration.

```
tuner = kt.Hyperband(model_tu,
                     objective='val_accuracy',
                     max_epochs=20,
                     factor=3,
                     directory='/kaggle/working',
                     project_name='rsna_cancer')
tuner.search(train_ds, validation_data = validation_ds, epochs=50,
                 callbacks=[stop_early])

# Get the optimal hyperparameters
best_hps=tuner.get_best_hyperparameters(num_trials=1)[0]

print(f"""The hyperparameter search is complete. The optimal number of units
in the optimized densely-connected layer is {best_hps.get('units')} and
the optimal learning rate for the optimizer is {best_hps.get('learning_rate')}.""")

#Output

The hyperparameter search is complete. The optimal number of units in the optimized
densely-connected layer is 480 and the optimal learning rate for the optimizer is 1e-05.
```




```
model = tuner.hypermodel.build(best_hps)
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 resnet50v2 (Functional)     (None, 2048)              23564800

 dense_5 (Dense)             (None, 512)               1049088

 dense_6 (Dense)             (None, 480)               246240

 dense_7 (Dense)             (None, 256)               123136

 dense_8 (Dense)             (None, 128)               32896

 dropout_1 (Dropout)         (None, 128)               0

 dense_9 (Dense)             (None, 1)                 129

=================================================================
Total params: 25,016,289
Trainable params: 1,451,489
Non-trainable params: 23,564,800
_________________________________________________________________
```

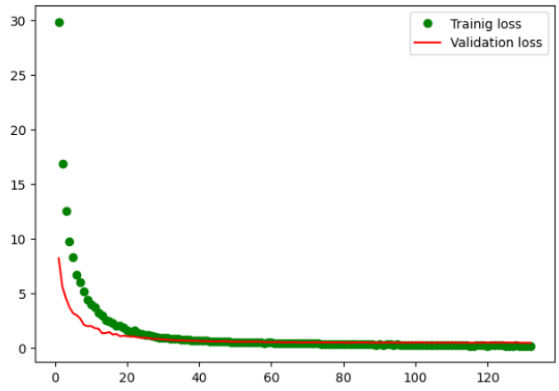
```
history = model.fit(train_ds, validation_data = validation_ds, epochs = 140,
                                        verbose = 1, workers = 8)

val_acc_per_epoch = history.history['val_accuracy']
best_epoch = val_acc_per_epoch.index(max(val_acc_per_epoch)) + 1
print('Best epoch: %d' % (best_epoch,))

#Output

Best epoch: 132
```

```
hypermodel = tuner.hypermodel.build(best_hps)
history = hypermodel.fit(train_ds, validation_data = validation_ds, epochs = best_epoch)
```

1.4.3 Results

The Model classify the training image with the accuracy of 0.9367 and classify unseen images during training with the 0.8683 accuracy after 132 epochs training.

```
Epoch 121/132
55/55 [==============================] - 3s 48ms/step - loss: 0.2011 - accuracy: 0.9136 - val_loss: 0.4548 - val_accuracy: 0.863
9
Epoch 122/132
55/55 [==============================] - 3s 48ms/step - loss: 0.1908 - accuracy: 0.9229 - val_loss: 0.4569 - val_accuracy: 0.872
6
Epoch 123/132
55/55 [==============================] - 3s 49ms/step - loss: 0.2038 - accuracy: 0.9177 - val_loss: 0.4680 - val_accuracy: 0.866
1
Epoch 124/132
55/55 [==============================] - 3s 48ms/step - loss: 0.2118 - accuracy: 0.9263 - val_loss: 0.4563 - val_accuracy: 0.859
6
Epoch 125/132
55/55 [==============================] - 3s 49ms/step - loss: 0.1725 - accuracy: 0.9401 - val_loss: 0.4568 - val_accuracy: 0.859
6
Epoch 126/132
55/55 [==============================] - 3s 48ms/step - loss: 0.1726 - accuracy: 0.9338 - val_loss: 0.4684 - val_accuracy: 0.863
9
Epoch 127/132
55/55 [==============================] - 4s 70ms/step - loss: 0.1878 - accuracy: 0.9252 - val_loss: 0.4525 - val_accuracy: 0.863
9
Epoch 128/132
55/55 [==============================] - 3s 50ms/step - loss: 0.1926 - accuracy: 0.9257 - val_loss: 0.4724 - val_accuracy: 0.866
1
Epoch 129/132
55/55 [==============================] - 3s 49ms/step - loss: 0.1770 - accuracy: 0.9326 - val_loss: 0.4447 - val_accuracy: 0.855
3
Epoch 130/132
55/55 [==============================] - 3s 48ms/step - loss: 0.1793 - accuracy: 0.9315 - val_loss: 0.4468 - val_accuracy: 0.863
9
Epoch 131/132
55/55 [==============================] - 3s 50ms/step - loss: 0.1644 - accuracy: 0.9436 - val_loss: 0.4403 - val_accuracy: 0.855
3
Epoch 132/132
55/55 [==============================] - 3s 49ms/step - loss: 0.1685 - accuracy: 0.9367 - val_loss: 0.4394 - val_accuracy: 0.868
3
```



Chapter 2

Contradictory Text Analysis

2.1 Context

The Contradictory, My Dear Watson dataset is a collection of text examples specifically designed to capture and showcase instances of contradictions in natural language. It serves as a valuable resource for training and evaluating models in the field of Natural Language Processing (NLP) that focus on detecting and resolving contradictions in text.
We can create deep learning models with Contradictory, My Dear Watson dataset to classify the sentences. Given a sentence pair (a **premise** and a **hypothesis**) there are 3 ways that they could be related:

1. **Entailment**, one sentence entails the other.
2. **Neutral**, the sentences are neutral but related.
3. **Contradiction**, one sentence contradicts the other.

The applications of the Contradictory, My Dear Watson dataset are numerous. NLP models trained on this dataset can be utilized in various domains, such as news aggregation platforms, social media analysis and question answering systems.

2.2 Project Objectives

The project's overarching aim will be accomplished through the following list of objectives:

1. Preprocessing of sequential dataset for a Deep Learning Model.
2. Implementing a Deep Learning Model for text analysis of sequential dataset.
3. Implementing a Deep Learning Model to identify and analyze a relation like contradiction, Neutral and Entailment in textual data.

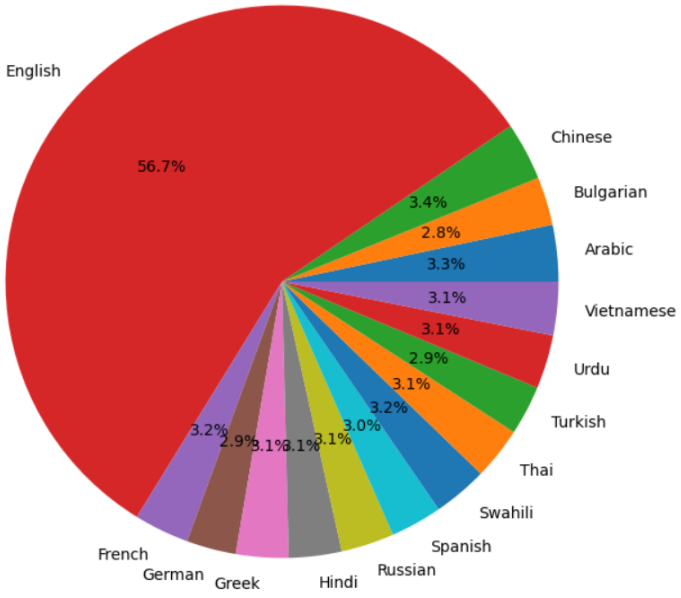
2.3 Data description

We have dataset “contradictory-my-dear-watson”. This data set consists of over 17K samples with over 12K training samples and over 5K test samples. This dataset contains the sentences of many languages but we will analysis texts in English languages with different labels.

This dataset have six columns with id, premise, hypothesis, lang-abv, language and label.

	id	premise	hypothesis	lang_abv	language	label
0	5130fd2cb5	and these comments were considered in formulat...	The rules developed in the interim were put to...	en	English	0
1	5b72532a0b	These are issues that we wrestle with in pract...	Practice groups are not permitted to work on t...	en	English	2
2	3931fbe82a	Des petites choses comme celles-là font une di...	J'essayais d'accomplir quelque chose.	fr	French	0
3	5622f0c60b	you know they can't really defend themselves l...	They can't defend themselves because of their ...	en	English	0
4	86aaa48b45	ในการเล่นบทบาทสมมุติก็เช่นกัน โอกาสที่จะได้แสด...	เด็กสามารถเห็นได้ว่าชาติพันธุ์แตกต่างกันอย่างไร	th	Thai	1

In this dataset premise and hypothesis are sentences in different languages. Model will be build to analysis the relation between these pairs of sentences with label 0,1 and 2. Here 0,1 and 2 show that the sentences contradicts, entails and neutral respectively. In the next figure we can see the percentage of samples of different languages in the dataset.



2.4 Data Preprocessing

As we have seen that dataset contains many languages but we will consider only English

language sentences.

```
train = train.drop(train[~(train['lang_abv'] == 'en')].index)
validation = validation.drop(validation[~(validation['lang_abv'] == 'en')].index)
validation_labels = validation_labels[validation_labels['id'].isin(validation['id'])]
display(train.head())
display(validation.head())
display(validation_labels.head())
```

At this stage, in the dataset still we have some unnecessary columns like id, lang-abv and language. We will remove these columns in the data frame and reset the index.

```
train = train.drop(columns=['id', 'lang_abv', 'language'])
validation = validation.drop(columns=['id', 'lang_abv', 'language'])
validation_labels = validation_labels.drop(columns=['id'])
validation = validation.reset_index()
validation = validation.drop(columns=['index'])
display(validation.head())

validation_labels = validation_labels.reset_index()
validation_labels = validation_labels.drop(columns=['index'])
display(validation_labels.head())
```

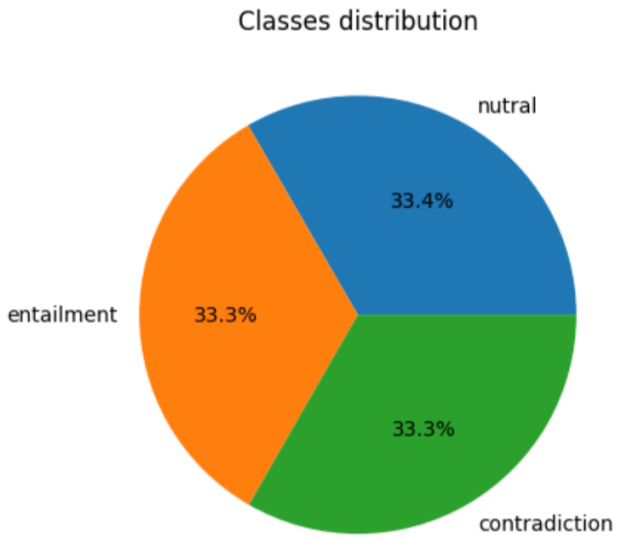
Due to removing the other languages the dataset becomes short. We need a big dataset so that our model can learn well. That's why we will consider another dataset namely Xnli which can help to train the model. We are considering this dataset because this dataset actually have same description as original dataset which we have already prepossessed. First we will convert Xnli dataset into data frame and then we concatenate this dataset with original dataset.

```
xnli_dataset = load_dataset('xnli','en')
# Access the train split of the XNLI dataset
xnli_train = xnli_dataset['train']
xnli_train =pd.DataFrame(xnli_train)
display(xnli_train.head())
# Access the validation split of the XNLI dataset
xnli_validation = xnli_dataset['validation']
xnli_validation =pd.DataFrame(xnli_validation)
display(xnli_validation.head())
# Access the test split of the XNLI dataset
```

```
xnli_test = xnli_dataset['test']
xnli_test =pd.DataFrame(xnli_test)
display(xnli_test.head())
xnli_validation['label'].value_counts()
xnli_df= pd.concat([data_set,xnli_train,xnli_validation,xnli_test])
display(xnli_df.head())
display(xnli_df.shape)
```

Dataset in Xnli is equally labeled that's why when it concatenates with original dataset it influences to equally classify obtaining dataset.

```
# Define the data
labels = ['nutral', 'entailment', 'contradiction']
values = xnli_df['label'].value_counts()
# Create the pie chart
plt.pie(values, labels=labels)
# Add a title
plt.title('Class distribution')
# Show the plot
plt.show()
total = sum(values)
percentages = [(value / total) * 100 for value in values]
# Create pie chart with percentage values
fig, ax = plt.subplots()
ax.pie(values, labels=labels, autopct='%1.1f%%')
ax.set_title('Classes distribution')
```



Now we split dataset into training and test datasets with 42 random state and 20 percent test dataset.

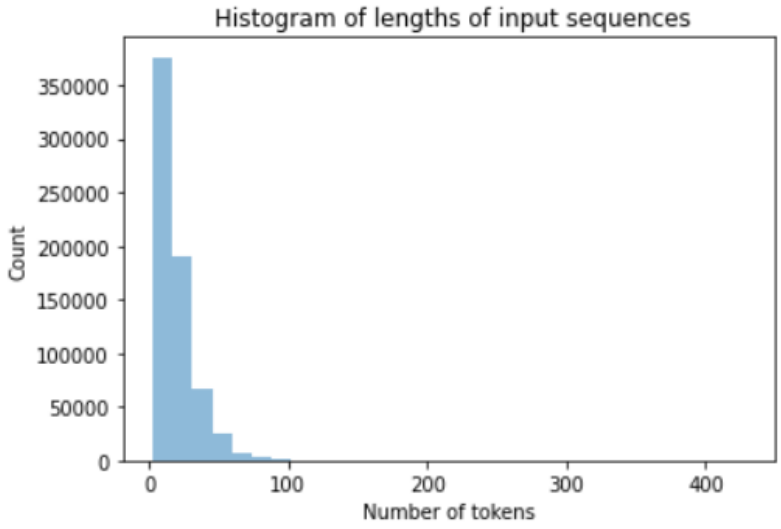
```
train, test = train_test_split(xnli_df,
                               stratify=xnli_df.label.values,
                               random_state=42,
                               test_size=0.2,
                               shuffle=True)

train.reset_index(drop=True, inplace=True)
test.reset_index(drop=True, inplace=True)
```

In many NLP tasks, text is represented as numerical features that deep learning models can understand. Tokenization enables the conversion of text into a sequence of tokens, which can then be transformed into numerical representations. The choice of BERT tokenizer is made to reduce the vocabulary size and improve generalization by treating lowercase and uppercase versions of the same word as equivalent. The BERT-uncased tokenizer typically performs the word splitting, special tokens, token IDs and attention masks. We will use BERT wordpiecetokenizer that performs like BERT-uncased tokenizer.

```
model_class = TFBertModel
tokenizer_class = BertTokenizer
model_name ='bert-base-uncased'

tokenizer = BertTokenizer.from_pretrained(model_name, num_labels=3)

save_path= '.'
if not os.path.exists(save_path):
    os.makedirs(save_path)
tokenizer.save_pretrained(save_path)
tokenizer = BertWordPieceTokenizer("vocab.txt", lowercase=True,
                   strip_accents=True, handle_chinese_chars=False)

tokenizer
```

Now we tokenize dataset based on BERT wordpiecetokenizer. We also visualize the tokenized data. In the following figure we can see that the most of tokenized sequences have length between 10 and 30.

```
def plot(df, tokenizer):
    all_text = df.premise.values.tolist() + df.hypothesis.values.tolist()
    all_text_tokenized = tokenizer.encode_batch(all_text)
    all_tokenized_len = [len(encoding.tokens) for encoding in all_text_tokenized]

    plt.hist(all_tokenized_len, bins=30, alpha=0.5)
    plt.title(' Histogram of lengths of input sequences')
    plt.xlabel('Number of tokens')
    plt.ylabel('Count')

    plt.show()

plot(train, tokenizer)
```



Before training a neural network model, we set hyperparameters which have following setting:

```
EPOCHS = 100
BATCH_SIZE = 256
MAX_LEN = 100
PATIENCE = 5
```

```
LEARNING_RATE = 1e-5
```

Deep learning models typically operate on numerical data. By encoding the tokenizer, we convert text into numerical representations that can be fed into these models for further processing and analysis. Encoding the tokenizer allows us to extract meaningful features from text data. These features capture important information about the text, such as the presence of specific words, word frequencies, or contextual relationships between words. These features can then be used as input to train the deep learning models. Each token or word is mapped to a numerical vector, which represents its position in a high-dimensional space. This vector space representation enables mathematical operations and computations on text data, such as measuring similarity between texts or performing vector arithmetic. Next we encode sequences of listed data. There are several popular methods for encoding sequences in NLP but we will use the following. Transformer-based Models: Transformer models, such as BERT (Bidirectional Encoder Representations from Transformers) has revolutionized many NLP tasks. This model use attention mechanisms to capture the contextual relationships between words in a sequence. It provide contextualized word embeddings, allowing the model to understand the meaning of words based on their surrounding context. The following code will give us three inputs namely input_word_ids, input_mask and input_type_ids. Thsese values we will use as input in our model.

```

def encode(df, tokenizer, max_len=100):
    pairs = df[['premise', 'hypothesis']].values.tolist()

    #check "BertTokenizer" do_lower_case = true
    tokenizer.enable_truncation(max_len)
    tokenizer.enable_padding()

    enc_list = tokenizer.encode_batch(pairs)
    input_word_ids = tf.ragged.constant([enc.ids for enc in enc_list], dtype=tf.int32)
    input_mask = tf.ragged.constant([enc.attention_mask for enc in enc_list],
                                dtype=tf.int32)
    input_type_ids = tf.ragged.constant([enc.type_ids for enc in enc_list],
                                dtype=tf.int32)

    inputs = {
        'input_word_ids': input_word_ids.to_tensor(),
        'input_mask': input_mask.to_tensor(),
        'input_type_ids': input_type_ids.to_tensor()}

    return inputs
```

```

train_input = encode(train, tokenizer=tokenizer, max_len=MAX_LEN)
train_ids = train_input['input_word_ids']
train_mask = train_input['input_mask']
train_type = train_input['input_type_ids']
train_labels = train.label.values
```

```
test_input = encode(test, tokenizer=tokenizer, max_len=MAX_LEN)
test_ids = test_input['input_word_ids']
test_mask = test_input['input_mask']
test_type = test_input['input_type_ids']
test_labels = test.label.values
```

Loading and processing data in batches offers several advantages in NLP and deep learning models. Loading and processing data in batches allows you to operate on smaller subsets of the data at a time, reducing memory requirements. This is particularly important when dealing with datasets that cannot fit entirely in memory. Modern deep learning frameworks like TensorFlow is optimized

for vectorized operations. These frameworks can efficiently perform operations on multiple data points simultaneously when they are organized in batches. Vectorized operations take advantage of hardware optimizations and are typically much faster than operating on individual data points sequentially. Many hardware architectures, such as GPUs and TPUs, are designed to efficiently process large batches of data. These architectures often have parallel processing units that can operate on multiple data points simultaneously. By loading and processing data in batches, you can fully utilize the computational power of these specialized hardware accelerators. Here we will do exactly same to load and process dataset in batches to reduce workload.

```

def create_dataset(features, labels, batch_size=BATCH_SIZE, validation=False):
    AUTO = tf.data.experimental.AUTOTUNE
    dataset = tf.data.Dataset.from_tensor_slices((features, labels))
              .shuffle(len(features))
    if validation:
        dataset = dataset.batch(batch_size).prefetch(AUTO)
    else:
        dataset = dataset.repeat().batch(batch_size).prefetch(AUTO)
    return dataset

```

```
training_data = create_dataset((train_ids, train_mask, train_type), train_labels,
                           batch_size=BATCH_SIZE)
testing_data = create_dataset((test_ids, test_mask, test_type), test_labels,
                          batch_size=BATCH_SIZE, validation=True)

```

2.5 Architecture and Training

After preparing our dataset we move on to applying our pre trained model namely 'Bert Base uncased model'. Three input layers are defined: input_word_ids, input_mask, and input_type_ids. These layers represent the input tokens, attention mask, and input type IDs, respectively. The shapes and data types of these inputs are specified.

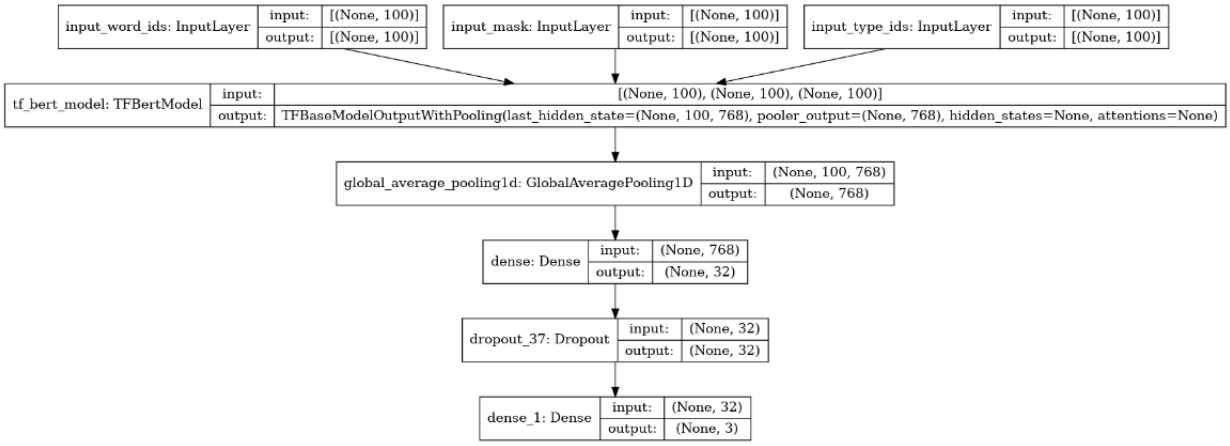
```

def build_model(model_name, model_class, max_len=50, add_input_type=False):
    tf.random.set_seed(123)

    encoder = model_class.from_pretrained(model_name)

    input_word_ids = tf.keras.Input(shape=(max_len,),
                                    dtype=tf.int32,
                                    name="input_word_ids")
    input_mask = tf.keras.Input(shape=(max_len,),
                                dtype=tf.int32,
                                name="input_mask")
    input_type_ids = tf.keras.Input(shape=(max_len,),
                                    dtype=tf.int32,
                                    name="input_type_ids")

    if add_input_type:
        features = encoder([input_word_ids, input_mask, input_type_ids])[0]
    else:
        features = encoder([input_word_ids, input_mask])[0]

    vector = tf.keras.layers.GlobalAveragePooling1D()(features)
    vector = tf.keras.layers.Dense(32, activation='relu')(vector)
    vector = tf.keras.layers.Dropout(0.5)(vector)

    output = tf.keras.layers.Dense(3, activation="softmax")(vector)

    if add_input_type:
        model = tf.keras.Model(inputs=[input_word_ids, input_mask, input_type_ids],
                               outputs=output)
    else:
        model = tf.keras.Model(inputs=[input_word_ids, input_mask],outputs=output)

    loss = tf.keras.losses.SparseCategoricalCrossentropy()
    optimizer = tf.keras.optimizers.Adam(learning_rate=LEARNING_RATE)
    model.compile(optimizer=optimizer,
                  loss=loss,
                  metrics=['accuracy'])
    return model
```

```

with strategy.scope():
```

```
    model = build_model(model_name,
                        model_class,
                        MAX_LEN,
                        add_input_type=True)
    model.summary()

```

```
Model: "model"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_word_ids (InputLayer)    [(None, 100)]        0           []

 input_mask (InputLayer)        [(None, 100)]        0           []

 input_type_ids (InputLayer)    [(None, 100)]        0           []

 tf_bert_model (TFBertModel)    TFBaseModelOutputWi  109482240   ['input_word_ids[0][0]',
                                thPoolingAndCrossAt               'input_mask[0][0]',
                                tentions(last_hidde               'input_type_ids[0][0]']
                                n_state=(None, 100,
                                 768),
                                 pooler_output=(Non
                                e, 768),
                                 past_key_values=No
                                ne, hidden_states=N
                                one, attentions=Non
                                e, cross_attentions
                                =None)

 global_average_pooling1d (Glob  (None, 768)         0           ['tf_bert_model[0][0]']
 alAveragePooling1D)

 dense (Dense)                  (None, 32)           24608       ['global_average_pooling1d[0][0]'
                                                                 ]

 dropout_37 (Dropout)           (None, 32)           0           ['dense[0][0]']

 dense_1 (Dense)                (None, 3)            99          ['dropout_37[0][0]']

==================================================================================================
Total params: 109,506,947
Trainable params: 109,506,947
Non-trainable params: 0
__________________________________________________________________________________________________
```

```
plot_model(model, to_file='model.png', show_shapes=True)
```



2.6 Results

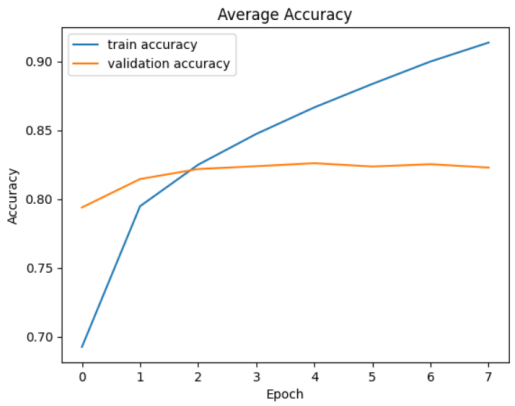
The Model classify the training data with the accuracy of 0.9135 and classify unseen data during training with the 0.8228 accuracy after 8 epochs training.

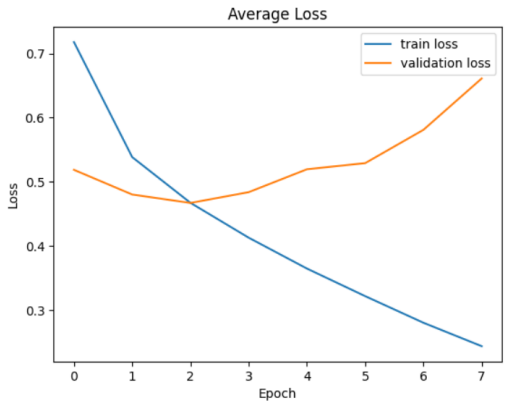
```
checkpoint_filepath='bert_best_checkpoint.hdf5'

callbacks = [EarlyStopping(monitor='val_loss',
                           mode='min',
                           verbose=1,
                           patience=PATIENCE,
                           min_delta=0.01),
             ModelCheckpoint(filepath=checkpoint_filepath,
                             save_best_only=True,
                             save_weights_only=True,
                             monitor='val_loss',
                             mode='min',
                             verbose=1)]

n_steps = int(train_ids.shape[0]/BATCH_SIZE)
train_history = model.fit(x=training_data,
                          validation_data=testing_data,
                          epochs=EPOCHS,
                          verbose=1,
                          steps_per_epoch=n_steps,
                          callbacks=callbacks)
```

```
Epoch 1: val_loss improved from inf to 0.51878, saving model to bert_best_checkpoint.hdf5
1272/1272 [==============================] - 279s 144ms/step - loss: 0.7178 - accuracy: 0.6926 - val_loss: 0.5188 - val_accuracy: 0.7938
Epoch 2/20
1272/1272 [==============================] - ETA: 0s - loss: 0.5386 - accuracy: 0.7947
Epoch 2: val_loss improved from 0.51878 to 0.48030, saving model to bert_best_checkpoint.hdf5
1272/1272 [==============================] - 168s 132ms/step - loss: 0.5386 - accuracy: 0.7947 - val_loss: 0.4803 - val_accuracy: 0.8145
Epoch 3/20
1272/1272 [==============================] - ETA: 0s - loss: 0.4674 - accuracy: 0.8249
Epoch 3: val_loss improved from 0.48030 to 0.46715, saving model to bert_best_checkpoint.hdf5
1272/1272 [==============================] - 167s 132ms/step - loss: 0.4674 - accuracy: 0.8249 - val_loss: 0.4671 - val_accuracy: 0.8217
Epoch 4/20
1272/1272 [==============================] - ETA: 0s - loss: 0.4130 - accuracy: 0.8472
Epoch 4: val_loss did not improve from 0.46715
1272/1272 [==============================] - 166s 131ms/step - loss: 0.4130 - accuracy: 0.8472 - val_loss: 0.4840 - val_accuracy: 0.8238
Epoch 5/20
1272/1272 [==============================] - ETA: 0s - loss: 0.3649 - accuracy: 0.8665
Epoch 5: val_loss did not improve from 0.46715
1272/1272 [==============================] - 166s 131ms/step - loss: 0.3649 - accuracy: 0.8665 - val_loss: 0.5197 - val_accuracy: 0.8260
Epoch 6/20
1272/1272 [==============================] - ETA: 0s - loss: 0.3219 - accuracy: 0.8835
Epoch 6: val_loss did not improve from 0.46715
1272/1272 [==============================] - 166s 131ms/step - loss: 0.3219 - accuracy: 0.8835 - val_loss: 0.5292 - val_accuracy: 0.8236
Epoch 7/20
1272/1272 [==============================] - ETA: 0s - loss: 0.2804 - accuracy: 0.8998
Epoch 7: val_loss did not improve from 0.46715
1272/1272 [==============================] - 166s 131ms/step - loss: 0.2804 - accuracy: 0.8998 - val_loss: 0.5811 - val_accuracy: 0.8252
Epoch 8/20
1272/1272 [==============================] - ETA: 0s - loss: 0.2439 - accuracy: 0.9135
Epoch 8: val_loss did not improve from 0.46715
1272/1272 [==============================] - 166s 131ms/step - loss: 0.2439 - accuracy: 0.9135 - val_loss: 0.6612 - val_accuracy: 0.8228
Epoch 8: early stopping
```

```python
def plot_acc(history):
    plt.plot(history.history['accuracy'], label='train accuracy')
    plt.plot(history.history['val_accuracy'], label='validation accuracy')
    plt.title('Average Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.show()

plot_acc(train_history)
```

```python
def plot_loss(history):
    plt.plot(history.history['loss'], label='train loss')
    plt.plot(history.history['val_loss'], label='validation loss')
    plt.title('Average Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()
plot_loss(train_history)
```






```
validation_input = encode(validation, tokenizer=tokenizer, max_len=MAX_LEN)
prediction = model.predict(validation_in
validation_predictions = [np.argmax(i) for i in model.predict(validation_input)]
validation['prediction'] = validation_predictions
validation_label.head()
```

```
sum = 0
size = validation.shape[0]
for i in range(size):
    sum = sum + (validation['prediction'][i] == validation_label['prediction'][i])
print("Correct:", sum)
print("Total:", size)
print("Accuracy:", sum/size)
```

```
Correct: 1061
Total: 2945
Accuracy: 0.3602716468590832
```