Trabajo Práctico 2: Clasificación bayesiana

Andrey Arguedas Espinoza	Heiner León Aguilar	Victor A Ortiz Ruiz
2020426569	2013006040	8705127
and12rx12@gmail.com	heiner@hey.com	voruiz@gmail.com

1. Introducción

Para este trabajo práctico realizamos una implementación del algoritmo de 'Naive Bayes' para el dataset MNIST.

El objetivo es que nuestro modelo logre clasificar correctamente imágenes de números dibujados a mano e identifique cual es el numero dibujado. Para esto utilizaremos 60 imágenes de cada numero para un total de 600 (números del 0 al 9), donde cada imagen contiene 784 píxeles los cuales "binarizaremos"para trabajar con valores de 0-1.

Figura 1: Ejemplo de imágenes de números en el dataset

2. Implementación

2.1. Binarización de la imagen

El primer paso es convertir los píxeles a posibles valores de 0 (apagado) y 1 (encendido)

```
def binarize_image(image_tensor):
    image_tensor[image_tensor > 0.5] = 1
    image_tensor[image_tensor <= 0.5] = 0
    return image_tensor
```

Figura 2: Método de binarización

2.2. Train model

Para esto crearemos una matriz de k filas (numero de clases, en este caso son 10) y 784 columnas(una para cada píxel) que poseerá las probabilidades de que un píxel sea 0 dado la clase k y luego calcularemos la matriz opuesta (para píxeles con valor 1)

```
def train_model(train_data_tensor, labels_training, num_classes = 10):
    labels_probabilities = get_labels_probabilities(labels_training)
    matrix_probabilities_1_given_k = generate_probabilities_matrix(train_data_tensor, labels_training, labels_probabilities, num_classes)
    matrix_probabilities_0_given_k = 1 - matrix_probabilities_1_given_k
    p_m_pix_val_given_k = [matrix_probabilities_0_given_k, matrix_probabilities_1_given_k]
    return (p_m_pix_val_given_k, labels_probabilities)
```

```
def get_labels_probabilities(labels_occurences):
  uniq, counts = labels_occurences.unique(return_counts=True)
  return torch.div(counts, labels_occurences.shape[0])

def generate_probabilities_matrix(train_data_tensor, labels_training, labels_probabilities, num_classes):
  complete_matrix = torch.zeros(num_classes, train_data_tensor.shape[0])
  general_probabilities_matrix = []

  for k in range(0, num_classes):
    k_type_observations = train_data_tensor[:, labels_training == k]
    k_pixels_probabilities = []
    sum = torch.sum(k_type_observations, 1)
    tensor_k_pixels_probabilities = torch.div(sum, k_type_observations.shape[1])
    complete_matrix[k].add_(tensor_k_pixels_probabilities)
  return complete_matrix
```

Finalmente nuestro modelo nos dará las 2 matrices en forma de lista para ser usados en el test.

2.3. Test model

Para poder testear el modelo tenemos que obtener las probabilidades de que un píxel m tome un valor dado las matrices de train, buscamos implementar la siguiente formula:

$$p\left(t_i = k|\overrightarrow{m}_i\right) \propto \prod_{d=0}^{D} p\left(m_{d,i}|t_i = k\right) p\left(t_i = k\right).$$

Figura 3: Formula

Sin embargo debido al problema del underflow nos vemos obligados a cambiar la multiplicatoria por una sumatoria de algoritmos, finalmente obtenemos la siguiente función:

```
def test_model(input_torch, p_m_pix_val_given_k, p_t_tensor, num_classes = 10):
    #assumes that the input comes in a row
    probs = torch.zeros(num_classes, input_torch.shape[0])
    idxsOnes = torch.nonzero(input_torch)
    idxsZeros = (input_torch == 0).nonzero()
    probs[:, idxsZeros] = torch.log(p_m_pix_val_given_k[0][:, idxsZeros])
    probs[:, idxsOnes] = torch.log(p_m_pix_val_given_k[1][:, idxsOnes])
    probs = probs + torch.log(p_t_tensor.view(num_classes, 1))
    scores_classes = torch.sum(probs, 1)
    return (torch.argmax(scores_classes).item(), scores_classes)
```

Figura 4: Test Model

Finalmente implementemos un test model por batches.

```
def test_model_batch(test_set, labels, p_m_pix_val_given_k, p_t_tensor):
    right_predictions = 0
    for image in range(0, test_set.shape[1]):
      (predicted_label, score_clasess) = test_model(test_set[:, image], p_m_pix_val_given_k, p_t_tensor, 10)
      if(predicted_label == labels[image]):
          right_predictions += 1
    return right_predictions / test_set.shape[1]
```

Figura 5: Test Model Batch

2.4. Resultado final para todo el dataset

```
(train_data_tensor, labels_tensor) = load_dataset(path = "/content/drive/MyDrive/Colab Notebooks/mnist_dataset/train")

p_m_pix_val_given_k , p_t_tensor = train_model(train_data_tensor, labels_tensor)

#Predecir el label de una observacion, la observacion #500
#(predicted_label, scores_classes) = test_model(train_data_tensor[:, 500], p_m_pix_val_given_k, p_t_tensor, 10)

accuracy = test_model_batch(train_data_tensor, labels_tensor, p_m_pix_val_given_k, p_t_tensor)

print("The accuracy is", accuracy)
```

```
The accuracy is 0.9166666666666666
```

Figura 6: Resultados Finales para todo el dataset

3. Experimentos

Para probar el modelo bayesiano entrenado con el dataset de MNIST, vamos a hacer dos experimentos: uno donde el dataset esta balanceado usando un 70 % de los datos para entrenar y el resto para pruebas, y otro con un dataset desbalanceado donde usamos un número de observaciones para entrenamiento y prueba según la clase.

3.1. Training y test set: 70/30

Para este experimento, se uso la función "train_test_split" de la biblioteca sklearn. Esta función nos permite dividir el dataset de entrenamiento en dos subsets: training y test set. Para este escenario, usamos un 70 % de los datos para entrenamiento y el resto (30 %) para probar el modelo.

Este experimento es realizado 10 veces y analizamos los resultados en fig. 7.

```
[ ] X = train_data_tensor.transpose(0,1)
    # Split the matrix and labels for training and test partitions
    # For reproductibility
    partitions_acc = []
    for i in range(0, 10) :
      X_train, X_test, y_train, y_test = train_test_split(X, labels_tensor, test_size=0.3)
      p_m_pix_val_given_k , p_t_tensor = train_model(X_train.transpose(0,1), y_train)
      accuracy = test_model_batch(X_test.transpose(0,1), y_test, p_m_pix_val_given_k, p_t_tensor)
      print('epoch :%s, accuracy: %s'%(i, accuracy))

      partitions_acc.append(accuracy)

    partitions_acc = np.array(partitions_acc)
    print('')
    print('partitions stats')
    print('accuracy mean: %s'%(partitions_acc.mean()))
    print('accuracy std: %s'%(partitions_acc.std()))
```

```
epoch :0, accuracy: 0.5388888888888889
epoch :1, accuracy: 0.45
epoch :2, accuracy: 0.5111111111111111
epoch :3, accuracy: 0.5055555555555555
epoch :4, accuracy: 0.46111111111111114
epoch :5, accuracy: 0.4166666666666667
epoch :6, accuracy: 0.5722222222222222
epoch :7, accuracy: 0.5166666666666667
epoch :8, accuracy: 0.4444444444444444
epoch :9, accuracy: 0.4166666666666667

partitions stats
accuracy mean: 0.4833333333333334
accuracy std: 0.050369086961917
```

Figura 7: Resultados del dataset balanceado

En promedio, el porcentaje de acierto rondó alrededor del 48.3334 % con una desviación estándar de 0.0583.

3.2. Tabla de particiones

El siguiente escenario vamos a evaluar el modelo usando un dataset desbalanceado según la siguiente tabla:

Clase	No. observaciones para test	No. observaciones para training
0	18	22
1	18	22
2	18	22
3	18	22
4	18	22
5	18	42
6	18	42
7	18	42
8	18	42
9	18	42

Para este escenario, primero tomamos el dataset y lo *barajamos* por cada partición. Luego sacamos el número específico de observaciones por cada clase como se puede ver la fig. 8.

```
[ ] X = train_data_tensor.transpose(0,1)

    # The numbers of observations per class, for this scenario.
    total_num_observations_per_class_train = [22, 22, 22, 22, 22, 42, 42, 42, 42, 42]
    total_num_observations_per_class_test = [18, 18, 18, 18, 18, 18, 18, 18, 18, 18]

    partitions_acc = []
    for i in range(0, 10) :
      # Shuffle the data before getting the observations
      indexes = torch.randperm(labels_tensor.shape[0])
      shuffle_X, shuffle_labels_tensor = X[indexes], labels_tensor[indexes]

      X_train, y_train, X_test, y_test = torch.Tensor([]), torch.Tensor([]), torch.Tensor([]), torch.Tensor([])

      num_observations_per_class_train = np.zeros(10, np.int8)
      num_observations_per_class_test = np.zeros(10, np.int8)

      # Create the training and test sets
      for j in range(0, len(shuffle_labels_tensor)):
        label = shuffle_labels_tensor[j]

        if num_observations_per_class_train[label] < total_num_observations_per_class_train[label]:
          if X_train.shape[0] == 0:
            X_train = torch.unsqueeze(shuffle_X[j], 0)
          else:
            X_train = torch.cat((X_train, torch.unsqueeze(shuffle_X[j], 0)), 0)

          y_train = torch.cat((y_train, torch.Tensor([shuffle_labels_tensor[j]])), 0)

          num_observations_per_class_train[label]+= 1

        elif num_observations_per_class_test[label] < total_num_observations_per_class_test[label]:
          if X_test.shape[0] == 0:
            X_test = torch.unsqueeze(shuffle_X[j], 0)
          else:
            X_test = torch.cat((X_test, torch.unsqueeze(shuffle_X[j], 0)), 0)

          y_test = torch.cat((y_test, torch.Tensor([shuffle_labels_tensor[j]])), 0)

          num_observations_per_class_test[label]+= 1
```

Figura 8: Código para obtener el dataset desbalanceado

Al igual que el caso anterior, se usaron 10 particiones diferentes y se obtuvo en promedio un porcentaje de 43.8889 % en aciertos con una desviación estándar de 0.0283.

```
    # Train the model with the partition
    p_m_pix_val_given_k , p_t_tensor = train_model(X_train.transpose(0,1), y_train)
    # Get the accuracy of the model
    accuracy = test_model_batch(X_test.transpose(0,1), y_test, p_m_pix_val_given_k, p_t_tensor)
    print('epoch :%s, accuracy: %s'%(i, accuracy))

    partitions_acc.append(accuracy)

partitions_acc = np.array(partitions_acc)
print('')
print('partitions stats')
print('accuracy mean: %s'%(partitions_acc.mean()))
print('accuracy std: %s'%(partitions_acc.std()))
```

```
epoch :0, accuracy: 0.4777777777777778
epoch :1, accuracy: 0.43333333333333335
epoch :2, accuracy: 0.38333333333333336
epoch :3, accuracy: 0.4444444444444444
epoch :4, accuracy: 0.4166666666666667
epoch :5, accuracy: 0.42777777777777776
epoch :6, accuracy: 0.4388888888888889
epoch :7, accuracy: 0.45
epoch :8, accuracy: 0.4888888888888889
epoch :9, accuracy: 0.42777777777777776

partitions stats
accuracy mean: 0.43888888888888894
accuracy std: 0.02832788618662658
```

Figura 9: Resultados del modelo usando el dataset desbalanceado

3.3. Comparación de resultados

El escenario con el dataset balanceado dio mejor porcentaje de aciertos debido a que cada clase contó con un mayor número de muestras para entrenamiento y pudo aprender de manera uniforme cada una de las clases.

En el caso del dataset desbalanceado, se redujo el número de muestras para entrenar las primeras 5 clases sin reducir el número de observaciones que se utilizaron para probar comparado a las clases con mayor número de muestras de entrenamiento. Esto definitivamente afecta la presición del modelo a encontrarse con datos que pertenece a las 5 clases con menos aprendizaje.

4. Conclusión

En este ejercicio pudimos ver la importancia de conocer como mejorar ciertos algoritmos mediante la aplicación de leyes matemáticas como los logaritmos para solucionar problemas de programación [1], además la importancia de implementar soluciones atómicas para luego extenderlas a problemas más complejos.Además de la importancia del dataset a la hora de entrenar el modelo.

Referencias

[1] Hong Zhang Kai Wang. A novel naive bayesian approach to inference with applications to the mnist handwritten digit classification. 2020.