

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines. Some nodes are highlighted with blue circles, and others with blue dots. The lines are thin and grey, creating a mesh-like structure.

CESTE | MiDS&AI

**Laboratorio de Proyectos**

# Casos Prácticos

Juan Badal • Jafet Benítez • José M. Calvo

A decorative network diagram in the bottom-right corner, similar to the one in the top-left. It shows a network of nodes and lines, with some nodes highlighted in blue. The overall style is clean and modern, with a focus on connectivity and structure.

# Sobre los proyectos

## Digit Recognizer

MNIST ("Modified National Institute of Standards and Technology") es el conjunto de datos "hola mundo" de facto de la visión artificial. Desde su lanzamiento en 1999, este conjunto de datos clásico de imágenes escritas a mano ha servido como base para comparar algoritmos de clasificación.

En esta competencia, su objetivo es identificar correctamente los dígitos de un conjunto de datos de decenas de miles de imágenes escritas a mano. Hemos seleccionado un conjunto de núcleos de estilo tutorial que cubren todo, desde la regresión hasta las redes neuronales. Lo alentamos a experimentar con diferentes algoritmos para aprender de primera mano qué funciona bien y cómo se comparan las técnicas.

## Natural Language Processing with Disaster Tweets

Twitter se ha convertido en un importante canal de comunicación en tiempos de emergencia.

La ubicuidad de los teléfonos inteligentes permite a las personas anunciar una emergencia que están observando en tiempo real. Debido a esto, más agencias están interesadas en monitorear Twitter mediante programación (es decir, organizaciones de socorro en casos de desastre y agencias de noticias).

Pero no siempre está claro si las palabras de una persona en realidad anuncian un desastre.

A decorative network diagram in the top-left corner, featuring a cluster of interconnected nodes. Some nodes are represented by solid grey circles, while others are open circles with a smaller solid circle inside. The nodes are connected by thin grey lines, forming a complex web-like structure.

1.

# Digit Recognizer

## Importación de los datos

```
import pandas as pd
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import MaxPooling2D
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Flatten
from tensorflow.keras.optimizers import RMSprop
from keras.preprocessing.image import ImageDataGenerator
from keras.callbacks import ReduceLROnPlateau

training_data = pd.read_csv('/kaggle/input/digit-recognizer/train.csv')
testing_data = pd.read_csv("/kaggle/input/digit-recognizer/test.csv")
```

## Comprobación de nulos

```
training = training_data.isnull().sum().sum()
print(f"Nulls in training: {training}")

validation = testing_data.isnull().sum().sum()
print(f"Nulls in validation: {validation}")
```

Nulls in training: 0

Nulls in validation: 0

## Data Preparation

- ⊙ Separación de X e Y para los datos de *train* y *test*
- ⊙ Normalización de los datos
- ⊙ Modificar forma de los datos
- ⊙ Dividir en datos de validación
- ⊙ Conversión a variables categóricas (one hot encoding)

## Separación de X e Y para los datos de *train* y *test*

```
train_X = training_data.iloc[:, 1:]  
  
train_Y = training_data.iloc[:, 0]  
  
test_X = testing_data  
  
train_X.shape, train_Y.shape
```

((42000, 784), (42000,))

## Normalización de los datos

```
train_X = train_X / 255.0  
test_X = test_X / 255.0
```



## Modificar forma de los datos

```
train_X = train_X.values.reshape(-1, 28, 28, 1)
test_X = test_X.values.reshape(-1, 28, 28, 1)

train_X.shape, test_X.shape
```

```
((42000, 28, 28, 1), (28000, 28, 28, 1))
```

## Dividir en datos de validación

```
train_X, val_X, train_Y, val_Y = train_test_split(train_X, train_Y, test_size=0.1)  
  
train_X.shape, val_X.shape, train_Y.shape, val_Y.shape
```

```
((37800, 28, 28, 1), (4200, 28, 28, 1), (37800,), (4200,))
```

## Conversión a variables categóricas (one hot encoding)

```
train_Y = keras.utils.to_categorical(train_Y, num_classes=10)  
val_Y = keras.utils.to_categorical(val_Y, num_classes=10)  
  
train_Y.shape, val_Y.shape
```

```
((37800, 10), (4200, 10))
```




## Modelo inicial

Para nuestra primera 'submission', apostamos por un modelo sencillo basado en una red neuronal simple

```
input_shape = [28*28]
model = keras.Sequential([
    layers.Dense(512, input_shape=input_shape),
    layers.Dense(128, activation='sigmoid'),
    layers.Dense(64, activation='relu'),
    layers.Dense(10)
])

model.compile(
    optimizer='adam',
    loss='mae'
)
```



## Modelo definitivo

Tras probar e investigar, decidimos que el camino a seguir para mejorar la precisión era una red neuronal convolucional (CNN)

```
def convolutional_model():
    input_img = tf.keras.Input(shape=(28,28,1))
    Z1 = tf.keras.layers.Conv2D(filters = 32, kernel_size = (5,5), strides=(1, 1), padding='same')(input_img)
    Z1 = tf.keras.layers.BatchNormalization(axis = 3) (Z1, training = True)
    A1 = tf.keras.layers.ReLU()(Z1)
    P1 = tf.keras.layers.MaxPool2D(pool_size = (2,2), strides = 2, padding = 'same')(A1)
    D01 = tf.keras.layers.Dropout(0.1)(P1)

    Z2 = tf.keras.layers.Conv2D(filters = 32, kernel_size = (3,3), strides=(1, 1), padding='same')(D01)
    Z2 = tf.keras.layers.BatchNormalization(axis = 3) (Z2, training = True)
    A2 = tf.keras.layers.ReLU()(Z2)
    P2 = tf.keras.layers.MaxPool2D(pool_size = (2,2), strides = 2, padding = 'same')(A2)
    D02 = tf.keras.layers.Dropout(0.1)(P2)

    Z3 = tf.keras.layers.Conv2D(filters = 64, kernel_size = (3,3), strides=(1, 1), padding='same')(D02)
    Z3 = tf.keras.layers.BatchNormalization(axis = 3) (Z3, training = True)
    A3 = tf.keras.layers.ReLU()(Z3)
    P3 = tf.keras.layers.MaxPool2D(pool_size = (2,2), strides = 2, padding = 'same')(A3)
    D03 = tf.keras.layers.Dropout(0.1)(P3)

    F1 = tf.keras.layers.Flatten()(D03)
    D1 = tf.keras.layers.Dense(256, activation='relu')(F1)
    D04 = tf.keras.layers.Dropout(0.1)(D1)

    F = tf.keras.layers.Flatten()(D04)

    outputs = tf.keras.layers.Dense(units = 10, activation = 'softmax') (F)

    model = tf.keras.Model(inputs=input_img, outputs=outputs)
    return model
```

```
model = convolutional_model()
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.summary()
```


```
early_stopping = keras.callbacks.EarlyStopping(
    patience=4,
    min_delta=0.001,
    restore_best_weights=True,
)

history = model.fit(datagen.flow(train_X, train_Y),
                    epochs=30,
                    validation_data=(val_X, val_Y),
                    batch_size=128,
                    callbacks=[early_stopping])
```



## Optimización del modelo (1/2)

Trabajamos los siguientes parámetros internos del modelo para tratar de mejorar su 'performance'

- ⦿ Número de capas que contiene la red
  - ⦿ Función de activación de las capas ocultas
  - ⦿ Función de activación de la capa de salida
  - ⦿ Tamaño del kernel
  - ⦿ Optimizador del modelo
  - ⦿ Dropout
  - ⦿ Batch normalization
- 

## Optimización del modelo (2/2)

Existen otras variables externas al modelo que han influido en su funcionamiento

- ⊙ Cantidad de datos disponibles para el entrenamiento (Train-Test split)

```
# split data into train and test slices

train_X, val_X, train_Y, val_Y = train_test_split(train_X, train_Y, test_size=0.1)

train_X.shape, val_X.shape, train_Y.shape, val_Y.shape
```

- ⊙ Data augmentation

```
# data augmentation

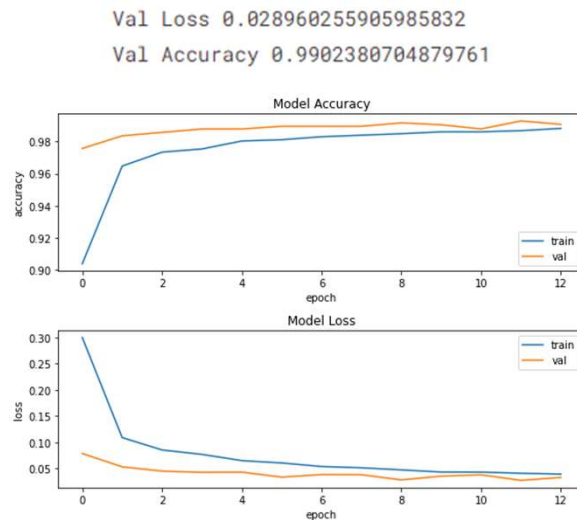
datagen = ImageDataGenerator(zoom_range = 0.1,
                             height_shift_range = 0.1,
                             width_shift_range = 0.1,
                             rotation_range = 10)

datagen.fit(train_X)
```

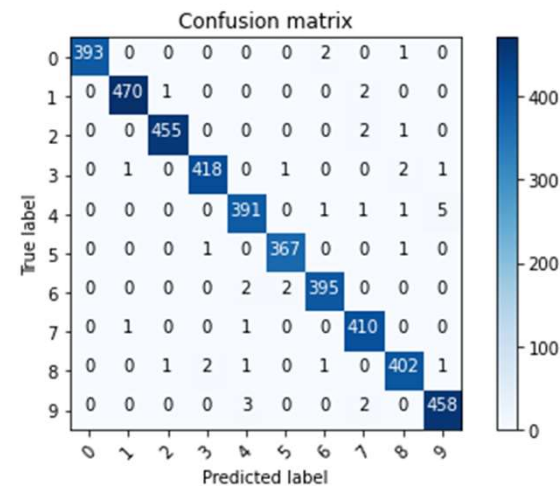
## Evaluación del modelo

Definimos las siguientes métricas y mecanismos para poder comprobar la efectividad del modelo

### ⦿ Métricas y 'plotting' (accuracy/loss)



### ⦿ Matriz de confusión





## Predicciones y comprobación del modelo

Realizamos las predicciones a través del modelo y estructuramos un mecanismo de testeo manual para comprobar el funcionamiento

### ⦿ Predicciones

```
# predict results
submissionPredictions = model.predict(test_X)

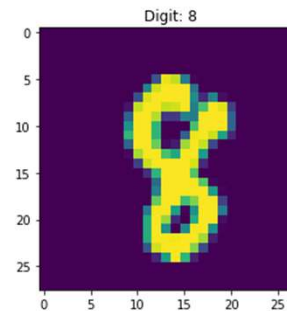
# select the index with the maximum probability
submissionPredictions = np.argmax(submissionPredictions, axis = 1)

submissionPredictions = pd.Series(submissionPredictions, name="Label")
```

### ⦿ Testeo manual

```
# testing results

itemToTest = 850
plt.imshow(test_X[itemToTest].reshape(28, 28))
# the label of the first number
plt.title(f"Digit: {submissionPredictions[itemToTest]}")
plt.show()
```





2.

# Natural Language Processing with Disaster Tweets

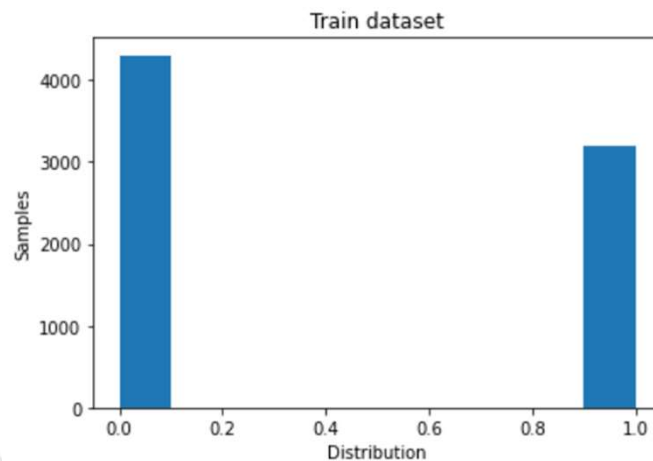
## Importación de los datos

```
1 import numpy as np
2 import pandas as pd
3 from sklearn import feature_extraction, linear_model, model_selection, preprocessing
4
5
6 dtype = {
7     'id': 'uint64',
8     'keyword': 'object',
9     'location': 'category',
10    'text': 'object',
11    'target': 'uint64',
12 }
13
14 train_df = pd.read_csv("/kaggle/input/nlp-getting-started/train.csv", dtype=dtype)
15 test_df = pd.read_csv("/kaggle/input/nlp-getting-started/test.csv", dtype=dtype,)
```

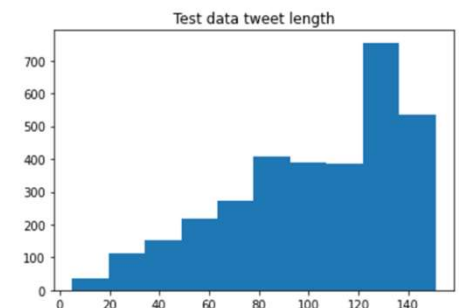
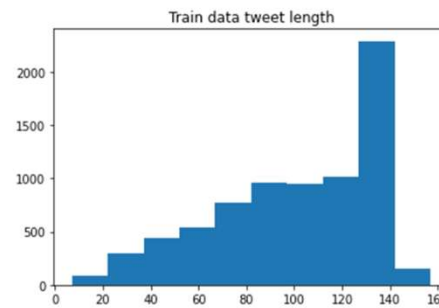
## Exploración y visualización de los datos (1/2)

Realizamos varias comprobaciones con el dataset para tratar de encontrar patrones o problemas en los datos que nos ayuden a desarrollar el modelo

- Distribución de tweets por etiqueta en el dataset de entrenamiento



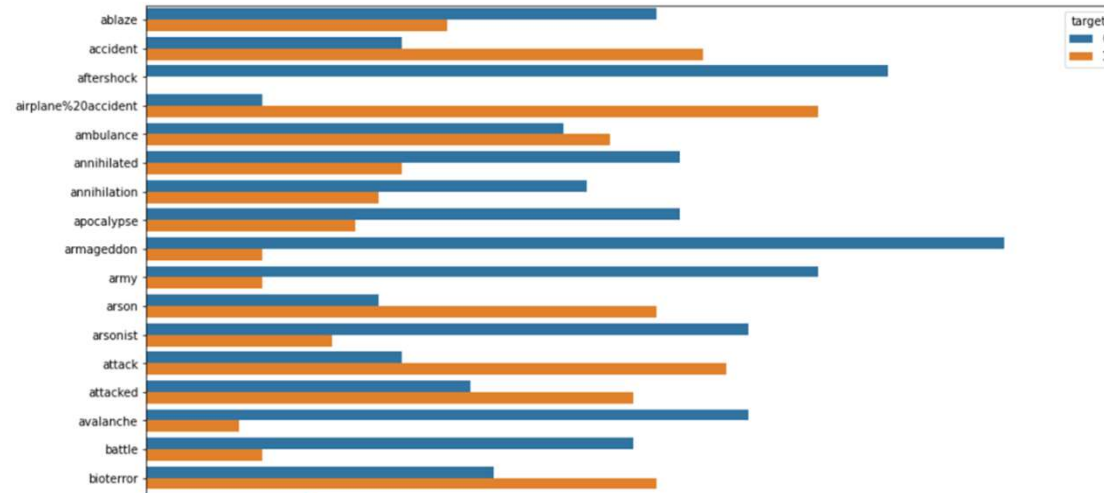
- Gráfico sobre la longitud de los tweets en ambos datasets



## Exploración y visualización de los datos (2/2)

Realizamos varias comprobaciones con el dataset para tratar de encontrar patrones o problemas en los datos que nos ayuden a desarrollar el modelo

- Exploración de la columna 'keyword' para ver cómo se distribuye cada una entre tweets con distinta etiqueta



## Preprocesamiento y preparación de los datos

```
8 def prepare_dataset(df):
9
10     df.loc[:, ['keyword']] = df['keyword'].str.replace('%20', ' ')
11
12     df['hashtags'] = df['text'].str.extract(regex)
13     df.loc[df['hashtags'].isna(), 'hashtags'] = ''
14
15     df.apply(fill_up_with_hashtag, axis=1)
16     df.loc[df['keyword'].isna(), 'keyword'] = ''
17
18     # Define vectorizer
19     vectorizer = feature_extraction.text.TfidfVectorizer(
20         stop_words='english',
21         strip_accents='unicode',
22         lowercase=True)
23
24     # Vectorize text column
25     X_text = vectorizer.fit_transform(df["text"])
26
27     # Clean and Vectorize keyword column
28     X_keyword = vectorizer.fit_transform(df['keyword'].values.astype('U'))
29
30     # Clean and Vectorize hashtags column
31     X_hashtags = vectorizer.fit_transform(df['hashtags'].values.astype('U'))
32
33     X = scipy.sparse.hstack((X_text,
34                             X_keyword,
35                             X_hashtags)).tocsr()
36
37     return X
```

Elimina los espacios en formato URL.

Extrae *hashtags*.

Recorre el *dataset* para rellenar el campo "keyword" en caso de que no se haya extraído algún *hashtag*.

Vectorizar variables de texto.

Agrupar los vectores relevantes en un sólo objeto.

## Modelo inicial

Para nuestra primera 'submission', apostamos por un modelo sencillo basado en SGDClassifier.

```
1 from sklearn.model_selection import train_test_split
2
3 X_train, X_test, y_train, y_test = train_test_split(
4     X, y,
5     test_size=0.33,
6     random_state=42)
7
8 ## Previously tested models
9 ## linear_model.RidgeClassifierCV() -> 0.7890
10 ## linear_model.RidgeClassifier() -> 0.7890
11 ## linear_model.SGDClassifier(max_iter=1500, tol=1e-3) -> 0.7906
12
13 clf = linear_model.SGDClassifier(max_iter=1500, tol=1e-3)
14 clf.fit(X_train, y_train)
```

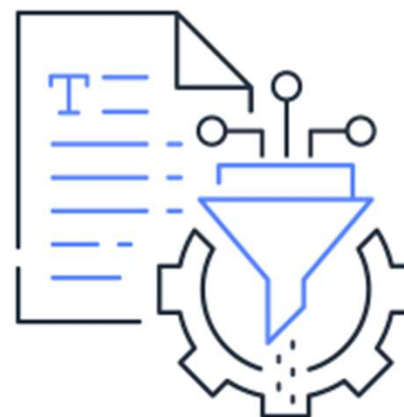
## Selección de modelo

Modelo	Precisión
RidgeClassifier	<b>78.90%</b>
SGDClassifier(max_iter=1500, tol=1e-3)	<b>79.06%</b>
LogisticRegression(max_iter = 2000, penalty = 'l2')	<b>79.41%</b>
XGBClassifier	<b>78.54%</b>
BernoulliNB	<b>81.21%</b>
GaussianNB	<b>61.75%</b>
ComplementNB	<b>79.88%</b>
DecisionTreeClassifier	<b>76.28%</b>
BERT	<b>87.34%</b>



## Sobre BERT

**BERT (Bidirectional Encoder Representations from Transformers)**, está diseñado para entrenar previamente representaciones bidireccionales profundas a partir de texto sin etiquetas al condicionar conjuntamente el contexto izquierdo y derecho en todas las capas. Como resultado, el modelo BERT pre-entrenado se puede ajustar con solo una capa de salida adicional para crear modelos de última generación para una amplia gama de tareas, como responder preguntas e inferencia de lenguaje, sin tareas sustanciales. modificaciones específicas de la arquitectura.



Fuente:

arXiv:1810.04805v2 [cs.CL] (2018), "[BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding](#)"

## Modelo definitivo

```
1 encoder_url = 'https://tfhub.dev/tensorflow/small_bert/bert_en_uncased_L-4_H-512_A-8/1'
2 preprocessor_url = 'https://tfhub.dev/tensorflow/bert_en_uncased_preprocess/3'
```

➔ Modelos pre-entrenados.

```
1 def build_bert_classifier():
2     tweet = tf.keras.layers.Input(shape=(), dtype=tf.string, name='tweets')
3     preprocessing_layer = hub.KerasLayer(preprocessor_url, name='preprocessing')
4     encoder_inputs = preprocessing_layer(tweet)
5     encoder = hub.KerasLayer(encoder_url, trainable=True, name='BERT_encoder')
6     outputs = encoder(encoder_inputs)
7     net = outputs['pooled_output']
8     net = tf.keras.layers.Dropout(0.1)(net)
9     net = tf.keras.layers.Dense(1, activation=None, name='classifier')(net)
10    return tf.keras.Model(tweet, net)
```

➔ Construcción del modelo.

```
1 loss = tf.keras.losses.BinaryCrossentropy(from_logits=True)
2 metrics = tf.metrics.BinaryAccuracy()
3 epochs = 4
4 steps_per_epoch = tf.data.experimental.cardinality(disaster_ds).numpy()
5 num_train_steps = steps_per_epoch * epochs
6 num_warmup_steps = int(0.1*num_train_steps)
7
8 init_lr = 3e-5
9 optimizer = optimization.create_optimizer(init_lr=init_lr,
10                                          num_train_steps=num_train_steps,
11                                          num_warmup_steps=num_warmup_steps,
12                                          optimizer_type='adamw')
13 bert_tweet_classifier.compile(optimizer=optimizer,
14                              loss=loss,
15                              metrics=metrics)
16 print(f'Training model with {encoder_url}')
17 history = bert_tweet_classifier.fit(x=disaster_ds,
18                                   epochs=epochs)
```

➔ Entrenamiento del modelo.

# Notebooks

## Caso 1 : Digit Recognizer

- ◎ [Digit Recognizer](#)
- ◎ [Digit CESTE \(4 capas, 0.1 dropout, no augmentation\)](#)
- ◎ [DigitRecognizer CESTE-CNN](#)
- ◎ [DigitRecognizer 4 layer CNN with data augmentation](#)

## Caso 2: Natural Language Processing with Disaster Tweets

- ◎ [Disaster Tweets CESTE 1.0](#)
- ◎ [NLP With Disaster Tweets - RidgeClassifier](#)
- ◎ [NLP With Disaster Tweets - SDGClassifier](#)
- ◎ [Disaster Tweets CESTE - Clasificadores](#)



**¡Gracias por su atención!**

**¿Preguntas?**

