```
%%capture
!pip install kaggle
```

U.S. Patent Phrase to Phrase Matching

Introduction

Natural Language Processing (NLP) is a field of AI that focuses on the interaction between computers and human languages. The goal of NLP is to enable machines to understand, interpret, and generate human language in a way that is both meaningful and useful.

This project entails fine tuning a **pretrained** NLP model using a library called `Huggingface Transformers.`

Tranformers are a type of NN architecture designed to handle sequential data, like text. It is the foundation of many modern NLP models. The open source library provides access to many pre-trained models. Hugging Face Transformers integrates perfectly with both *PyTorch* and *TensorFlow*.

A pretrained model has many parameters already fit. With *Universal Language Model Fine-tuning* (ULMFit) - not used here - the model is trained to predict the next word in a text. *Transformers* takes a different approach, training the model to predict the missing word in a document. We will be using the Transformers library in this project.

This type of training is called **self-supervised learning** as we do not need to give labels to our data, we just feed it lots of text. The labels are embedded in the independent variable. Self-supervised learning is not generally used for the model that is trained directly but instead for pretraining a model for **transfer learning**.

We can **fine-tune** the model to perform transfer learning, meaning we change the parameters we are not sure about to fit our purpose. When working with machine learning, many of the base parameters (such as detecting a 'corner' in an image) will be relevant in a wide range of applications (essentially all image classification tasks for example). By making use of an NLP pre-trained model, we can drastically reduce the amount of training required, as the model will have a good understanding of how language (and the world) works. We simply fine-tune it to understand the style of the corpus we are targetting.

It is also worth noting that the **encoder** is a term to describe the model without the task-specific final layer(s). It has the same meaning as the **body** whne describing vision CNNs but is used for NLP and generative models.

We will use the NLP model to match key phrases in United States patent documents for this Kaggle competition. It is crucial to understand what a **document** is: it is an input to an NLP model that contains text - its length is arbitrary.

We are tasked with comparing two phrases and scoring them whether they are similar or not, based on their patent class.

- Score of `1` means identical meanings.
- Score of `0` means completely different meanings.

We can have a score of `0.5` indicating a somewhat similar meaning with differences.

This isn't strictly a classification problem yet, as the answer can fall anywhere between 0 and 1. So, to convert it into a classification problem we will represent each piece of data as a question:

Which category does the following text fall under? `"TEXT1:{phrase1}; TEXT2:{phrase2}; CONTEXT:{context}"`

- Different
- Similar
- Identical

Collecting Data

Because this data comes from a Kaggle competion, we will have to collect from the Kaggle API.

```
import os

# now the Kaggle API knows where to look for the kaggle.json file with credenti
os.environ['KAGGLE_CONFIG_DIR'] = "/content"
```

```
%%capture
import zipfile, kaggle # to allow us to work with zip archives and interact wit
from pathlib import Path # to create and manipulate file paths in an OO manner

path = Path('us-patent-phrase-to-phrase-matching')
kaggle.api.competition_download_cli(str(path)) # download all files for a speci
zipfile.ZipFile(f'{path}.zip', 'r').extractall(path) # open ZIP file and extrac
```

We can now inspect the file.

```
!ls {path}
```

```
sample_submission.csv  test.csv  train.csv
```

Because we can see that the data is stored in CSV files, it will be ideal to use Pandas.

```
import pandas as pd

df = pd.read_csv(path/'train.csv')
df
```

	id	anchor	target	context	score
0	37d61fd2272659b1	abatement	abatement of pollution	A47	0.50
1	7b9652b17b68b7a4	abatement	act of abating	A47	0.75
2	36d72442aefd8232	abatement	active catalyst	A47	0.25
3	5296b0c19e1ce60e	abatement	eliminating process	A47	0.50
4	54c1e3b9184cb5b6	abatement	forest region	A47	0.00
...	...	...	...	...	...
36468	8e1386cbefd7f245	wood article	wooden article	B44	1.00
36469	42d9e032d1cd3242	wood article	wooden box	B44	0.50
36470	208654ccb9e14fa3	wood article	wooden handle	B44	0.50
36471	756ec035e694722b	wood article	wooden material	B44	0.75
36472	8d135da0b55b8c88	wood article	wooden substrate	B44	0.50

36473 rows × 5 columns

We can also use the `describe()` method to understand how each of the columns is used.

```
df.describe(include='object')
```

	id	anchor	target	context
count	36473	36473	36473	36473
unique	36473	733	29340	106
top	8d135da0b55b8c88	component composite coating	composition	H01
freq	1	152	24	2186

What we can note from the table above is that there is not much unique data to train our model with. There are only 733 unique anchors, with the most frequent 'component composite coating' being repeated 152 times.

Feature Engineering

Now we can use the suggestion that was made in the introduction to create a **series** in our dataframe that will be used as input to the model. A series is the 'pandas' name for a column. We are accessing each series using Python "dot" notation - but if we are altering the series, we should use the 'dictionary-type' notation instead.

Each entry in the input column represents a **document** which we will use to fine-tune the NLP model.

```
df['input'] = 'TEXT1: ' + df.anchor + ' TEXT2: ' + df.target + ' CONTEXT: ' + c
df.input
```

```
                                                input
0          TEXT1: abatement TEXT2: abatement of pollution...
1          TEXT1: abatement TEXT2: act of abating CONTEXT...
2          TEXT1: abatement TEXT2: active catalyst CONTEX...
3          TEXT1: abatement TEXT2: eliminating process CO...
4          TEXT1: abatement TEXT2: forest region CONTEXT:...
...                                                   ...
36468      TEXT1: wood article TEXT2: wooden article CONT...
36469      TEXT1: wood article TEXT2: wooden box CONTEXT:...
36470      TEXT1: wood article TEXT2: wooden handle CONTE...
36471      TEXT1: wood article TEXT2: wooden material CON...
36472      TEXT1: wood article TEXT2: wooden substrate CO...
36473 rows × 1 columns

dtype: object
```

Tokenization and Numericalization

Neural networks (NN) work with numbers. As we saw in the last notebook, available in the GitHub repository `frank-895/machine_learning_journey/tree/main/manual_creation_of_NN`. Each layer of the NN relies on matrix multiplication or activation functions - which we cannot do with strings.

As such, we have to perform two tasks:

- **Tokenization**, which is the process of breaking down text into smaller pieces, called **tokens**. Tokens are often words, subwords, or characters, depending on the level of tokenization. The list of all the distinct tokens (the *levels* of a variable) will define the **vocabulary**. The level at which the text is broken is down is called the **granularity**.
- **Numericalization**, which is converting each token into a number by assigning each token in the vocabulary with a digit.

This process depends on tokenization model we use and there are thousands available. If we use tokenisation that creates a vast vocabulary, our model may be very accurate but also very slow. The converse is true if our vocabulary is sparse.

Since we will be using Transformers for tokenization we need to convert our Pandas dataframe into a Huggingface dataset.

```
%%capture
!pip install datasets
from datasets import Dataset, DatasetDict
```

```
ds = Dataset.from_pandas(df)
ds
```

```
Dataset({
    features: ['id', 'anchor', 'target', 'context', 'score', 'input'],
    num_rows: 36473
})
```

Because we are using a pretrained model and fine-tuning it, we need to use the same tokenization mechanism as the pretrained model to ensure the vocabularies are identical. This means we do not need to make all the little decisions involved with effective tokenization.

We will choose our NLP model then use `AutoTokenizer` to create a tokenizer that is appropriate for our given model. AutoTokenizer is essentially just a dictionary which maps each model to a tokenizer.

Hugging Face Models contains over 1 million pretrained NLP models. They have a variety of different architectures trained on a variety of different **coropuses** (collections of written texts).

We could choose one of many models pretrained for use on patents; however, we will opt for a more general model for the purpose of learning, `deberta-vs-small`.

```
%%capture
model_nm = 'microsoft/deberta-v3-small'

from transformers import AutoModelForSequenceClassification, AutoTokenizer
tokz = AutoTokenizer.from_pretrained(model_nm)
```

Let's see how the tokenizer works...

The `'_'` represents the start of each word (as there is a different meaning between `'my'` at the start of a word and in the middle of a word).

```
tokz.tokenize("Hello, my name is Frank and I'm practicing NLP classification")
```

```
['▁Hello',
 ',',
 '▁my',
 '▁name',
 '▁is',
 '▁Frank',
 '▁and',
 '▁I',
 "'",
 'm',
 '▁practicing',
 '▁NLP',
 '▁classification']
```

Now, we will perform numericalization using a simple function.

```
def tok_func(x): return tokz(x["input"])
```

Numericalization can be a costly process, so we will perform this in parallel on every row in our dataset using `map`. We will inspect one row of our dataset, which now contains a column called `'input_ids'` which is the tokenized and numericalized version of input.

The number represents the position in the vocabulary of each word in the string.

```
tok_ds = ds.map(tok_func, batched=True)
tok_ds[0]['input'], tok_ds[0]['input_ids']
```

```
Map: 100%                                36473/36473 [00:04<00:00, 5342.27 examples/s]
```

```
('TEXT1: abatement TEXT2: abatement of pollution CONTEXT: A47',
 [1,
  54453,
  435,
  294,
  47284,
  54453,
  445,
  294,
  47284,
  265,
  6435,
  20967,
  104917,
  294,
  336,
  5753,
```

Finalizing Data

Our model will also require labels for the input to enable classification to take place.

Tranformers relies on a column called `labels`, so we will rename our `score` column in the dataset.

```
tok_ds = tok_ds.rename_columns({'score':'labels'})
```

Overfitting

In the last notebook, I briefly introduced the critical concept of **overfitting**.

Overfitting occurs when a model learns the details and noise in the training data instead of generalising the patterns. This will impact its ability to make accurate predictions on unseen data. Overfitting generally occurs when there is:

1. **Exessive complexity** - too many layers or parameters.
2. **Too few training examples** - not enough to learn from.
3. **Too many epochs** - stops finding patterns.

To ensure the model does not experience overfitting, it is critical to keep 3 seperate sets, or 'groups' of data.

1. **Training set** - where the model learns from the data by identifying patterns, relationships and features. Parameters are updated based on training data using loss and SGD.
2. **Validation set** - used to tune hyperparamaters. After updating the model's parameters on the training set, its performance is tested on the validation set. If validation performance is poor compared to training performance, this indicates the model is overfitting.
3. **Test set** - used to evaluate the model's final performance. This dataset is not used during training or validation, providing a fair estimate of how the model will perform on completely unseen data.

These sets will not overlap - i.e., no data from the validation set is in the training set.

It is also worth noting that **underfitting** occurs when there is not enough complexity in the model to match the data.

As seen above, the Kaggle competition has already provided a test set but not a validation set.

For this project, we will use a random split of data; however, in many situations this will be an *unideal* way of generating a validation set. Selecting a suitable validation set can be a complex task.

Consider for example, a time series. Generally, you would use this data to predict the future. So, a random split of data is not going to accurately represent how you will use the model. Instead, it would make more sense to truncate the model and save 'future' data for your validation set.

There have been other examples too of poor validation sets. For example, in a Kaggle competition, when identifying the species of fish caught, many models overfit as they would detect the type of fish based on the boat in the picture. Because the test set contained completely different boats, the models (which appeared to be performing well based on the validation set) were terrible.

This is a really dangerous issue in real-life, as it can be difficult to detect overfitting. This can slowly reduce value for organisations.

This is why it is critical to also keep a test set. If you try heaps of different models, it is possible you will eventually find a model that works very well on the validation set purely by coincidence. This is an example of overfitting to the validation set! And, the only way to detect this is by keeping a test set.

So, we keep a validation set to keep the model from overfitting to the training data and a test set to keep us from overfitting to the validation set!

```
# NOTE - the test set does not have labels.
eval_df = pd.read_csv(path/'test.csv')
eval_df.describe()
```

	id	anchor	target	context
count	36	36	36	36
unique	36	34	36	29
top	4112d61851461f60	hybrid bearing	inorganic photoconductor drum	G02
freq	1	2	1	3

Now, we need to finalise our test dataset so we will be able to run it through the model once it is created.

```
eval_df['input'] = 'TEXT1: ' + eval_df.anchor + ' TEXT2: ' + eval_df.target + '
eval_ds = Dataset.from_pandas(eval_df).map(tok_func, batched=True)
```

```
Map: 100%                                    36/36 [00:00<00:00, 1315.81 examples/s]
```

```
# Kaggle has already created a test set for us
dds = tok_ds.train_test_split(0.25, seed=42) # keep 25% of the data for the val
dds
```

```
DatasetDict({
    train: Dataset({
        features: ['id', 'anchor', 'target', 'context', 'labels', 'input',
'input_ids', 'token_type_ids', 'attention_mask'],
        num_rows: 27354
    })
    test: Dataset({
        features: ['id', 'anchor', 'target', 'context', 'labels', 'input',
'input_ids', 'token_type_ids', 'attention_mask'],
        num_rows: 9119
    })
})
```

Selecting a Metric

The **metric** (or ideally *metrics*) is used by the human to evaluate the model's performance. In this situation, the Kaggle competition has already selected a metric - *the Pearson correlation coefficient*. Typically, for classification, we would use *accuracy* (i.e., was the model correct or incorrect), which doesn't quite align with this task for obvious reasons.

The metric is typically different to the value calculated by the loss function - as the loss function requires a gradient (accuracy does not have a gradient as it is either correct or incorrect).

Selecting a good metric can be difficult and people are prone to selecting a metric that's easy to measure but not representative of how well a model is performing. Also, consider Goodhart's law - "*When a measure becomes a target, it ceases to be a good measure*".

Metrics need to be a proxy for what we really care about - not just something that is easy to measure. Often, what we care about is not easy to measure so it is easy to resort to easier metrics. For example, if we want to measure crime - the easiest thing to measure is arrests. But, because of existing bias, arrests are not an accurate measure of the amount of crime.

Furthermore, we can not always know exactly what uesrs will experience, meaning it is difficult to predict what metrics will quantify the user's experience.

This is not to say that metrics are not useful, but it is important they are selected thoughtfully and that a range of quality metrics are chosen. Since this project is not for professional purposes, we can continue with our single metric - but this is an important point to make when considering organisational excellence.

Because AI is particularly good at optimising metrics, it is critical to ensure metrics are properly chosen.

The Pearson correlation coefficient is usually abbreviated with r. It is the most widely used measure of the degree of relationship between variables.

r varies between -1 and $+1$ where:

- -1 indicates perfect inverse correlation
- $+1$ indicates perfect positive correlation

It is worth noting that r is very sensitive to outliers as it uses the square of the difference, like many metrics.

NOTE - it is very important to visualise data. It can help with identifying outliers or other issues (like truncation). It also allows you to understand what an r value of 0.6 (for example) **actually means**.

We will report the metric after every epoch to evalute the model's performance. Tranformers expects metrics in dictionary format so the trainer knows what label to use.

```
import numpy as np

def corr_d(eval_pred):
  """Calculates pearson correlation coefficient and returns the metric in a dic

  preds = eval_pred.predictions
  labels = eval_pred.label_ids

  return {'pearson': np.corrcoef(preds, labels)[0][1]}
```

`eval_preds` is given by the Transformers model. eval_preds will be a dictionary containing two keys:

- `predictions` which is the model's raw output.
- `label_ids` which is the true labels from the validation dataset.

We pass these to `np.corrcoef` which returns an array:

`[[1, 0.5], [0.5, 1]]`

This is in case you were calculating `r` for multiple combinations of variables.

Hence, we use `[0][1]` to select a single floating point number to use as the peason correlation coefficient.

Training the Model

What we call a "learner" in `fastai` we call a "trainer" in Transformers.

```
from transformers import TrainingArguments, Trainer
```

We define some of our **hyperparameters**, which define how a neural network is trained and influence its performance.

Batch size is the number of rows we pass to the model at a time. Batch size determines the size of a **mini-batch**. You want a large batch size to increase speed - but too large, and you will face a memory error.

The **learning rate** is the most important hyperparameter but unlike fastai, Transformers does not provide a learning rate finder to help you choose a suitable value. *Instead, we rely on trial and error.* Generally, start with a smaller learning rate and double it until the model falls apart.

```
bs = 256 # batch size
epochs = 4
lr = 8e-5 # learning rate
```

Transformers uses an object from the `TrainingArguments` class to set the hyperparameters. There is a lot of boilerplate here that generally works fine for most models - the main hyperparameters are listed above. Unlike fastai, which is more high-level, we have more power to customise our model with Transformers.

```
args = TrainingArguments('outputs',
                         learning_rate=lr, # important
                         warmup_ratio=0.1,
                         lr_scheduler_type='cosine',
                         fp16=True,
                         eval_strategy="epoch",
                         per_device_train_batch_size=bs, # important
                         per_device_eval_batch_size=bs*2,
                         num_train_epochs=epochs, # important
                         weight_decay=0.01,
                         report_to='none')
```

Now, we can create our Trainer model.

`model` is equivalent to a `visionLearner` model from fastai.

- `AutoModelForSequenceClassification` will create a model that is appropriate for classifying sequences.
- `from_pretrained` will ensure the model takes from a pretrained model.
- `model_nm` is the name of the pretrained model (deberta-v3).
- `num_labels` defines the number of output labels (i.e., the number of categories each input could be assigned to). In this case, the model will be given a single score representing the similiarity of the two phrases. By making `num_labels = 1` we essentilly turn `AutoModelForSequenceClassification` into a regression problem.

```
model = AutoModelForSequenceClassification.from_pretrained(model_nm, num_labels

trainer = Trainer(model, # chosen pretrained model and architecture
                  args, # hyperparameters
                  train_dataset=dds['train'], # training data
                  eval_dataset=dds['test'], # validation data
                  processing_class=tokz, # tokenizer function used in the pretr
                  compute_metrics=corr_d # the metrics we want the model to spi
                  )
trainer.train()
```

[428/428 02:57, Epoch 4/4]

Epoch	Training Loss	Validation Loss	Pearson
1	No log	0.025408	0.795821
2	No log	0.021728	0.822649
3	No log	0.023084	0.829732
4	No log	0.022361	0.830237

```
TrainOutput(global_step=428, training_loss=0.027497095482371677, metrics=
{'train_runtime': 177.4626, 'train_samples_per_second': 616.558,
'train_steps_per_second': 2.412, 'total_flos': 691495225233480.0,
'train_loss': 0.027497095482371677, 'epoch': 4.0})
```

We ended up with a 0.83 correlation!

The model is very effective already at the first epoch with 0.8 correlation. The only reason it was possible to gain such a high correlation is because the model is pretrained and already has a good idea of how similar two words are. By fine-tuning it for our task, we have been able to improve the correlation further. It would not have been possible to train such an effective model with the minimal data we had, without using a pretrained model.

Testing the Model

Now that we have a Trainer model, we can use it much like we would a Learner model from fastai.

```
# prevents scientific notation and rounds to two decimal places in numpy
np.set_printoptions(precision=2, suppress=True)

preds = trainer.predict(eval_ds).predictions.astype(float)
preds
```

```
array([[ 0.59],
       [ 0.74],
       [ 0.48],
       [ 0.32],
       [-0.  ],
       [ 0.51],
       [ 0.49],
       [ 0.  ],
       [ 0.2 ],
       [ 1.11],
       [ 0.19],
       [ 0.21],
       [ 0.79],
       [ 0.79],
       [ 0.72],
       [ 0.4 ],
       [ 0.25],
       [-0.01],
       [ 0.72],
       [ 0.27],
       [ 0.39],
       [ 0.25],
       [ 0.08],
       [ 0.23],
       [ 0.55],
       [-0.05],
       [-0.06],
       [-0.05],
       [-0.05],
       [ 0.64],
       [ 0.32],
       [ 0.03],
       [ 0.68],
       [ 0.47],
       [ 0.39],
       [ 0.18]])
```

It's good to inspect our predictions because a clear problem presents itself! Predictions should not be < 0 or > 1. Later, we will be using sigmoid functions to fix this issue, but for now, we can just round out-of-bound predictions to 0 or 1 as necessary.

```
preds = np.clip(preds, 0, 1)
preds
```

```
array([[0.59],
       [0.74],
       [0.48],
       [0.32],
       [0.  ],
       [0.51],
       [0.49],
       [0.  ],
       [0.2 ],
       [1.  ],
       [0.19],
       [0.21],
       [0.79],
       [0.79],
       [0.72],
       [0.4 ],
       [0.25],
       [0.  ],
       [0.72],
       [0.27],
       [0.39],
       [0.25],
       [0.08],
       [0.23],
       [0.55],
       [0.  ],
       [0.  ],
       [0.  ],
       [0.  ],
       [0.64],
       [0.32],
       [0.03],
       [0.68],
       [0.47],
       [0.39],
       [0.18]])
```

Submission

Now, we can create a submission file and check our performance in the Kaggle competition!

```
import datasets

submission = datasets.Dataset.from_dict({
    'id':eval_ds['id'],
    'score':preds
})

submission.to_csv('submission.csv', index=False)
```

```
Creating CSV from Arrow format: 100%                1/1 [00:00<00:00, 45.11ba/s]
855
```

Conclusion

Thank you for reading my learning journey with Hugging Face Transformers and NLP!