Plagiarism Detection Model

Now that you've created training and test data, you are ready to define and train a model. Your goal in this notebook, will be to train a binary classification model that learns to label an answer file as either plagiarized or not, based on the features you provide the model.

This task will be broken down into a few discrete steps:

- Upload your data to S3.
- Define a binary classification model and a training script.
- Train your model and deploy it.
- Evaluate your deployed classifier and answer some questions about your approach.

To complete this notebook, you'll have to complete all given exercises and answer all the questions in this notebook.

> All your tasks will be clearly labeled **EXERCISE** and questions as **QUESTION**.

It will be up to you to explore different classification models and decide on a model that gives you the best performance for this dataset.

Load Data to S3

In the last notebook, you should have created two files: a `training.csv` and `test.csv` file with the features and class labels for the given corpus of plagiarized/non-plagiarized text data.

> The below cells load in some AWS SageMaker libraries and creates a default bucket. After creating this bucket, you can upload your locally stored data to S3.

Save your train and test `.csv` feature files, locally. To do this you can run the second notebook "2_Plagiarism_Feature_Engineering" in SageMaker or you can manually upload your files to this notebook using the upload icon in Jupyter Lab. Then you can upload local files to S3 by using `sagemaker_session.upload_data` and pointing directly to where the training data is saved.

In [2]:

```
import pandas as pd
import boto3
import sagemaker
```

In [3]:

```
"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
# session and role
sagemaker_session = sagemaker.Session()
role = sagemaker.get_execution_role()

# create an S3 bucket
bucket = sagemaker_session.default_bucket()
```

EXERCISE: Upload your training data to S3

Specify the `data_dir` where you've saved your `train.csv` file. Decide on a descriptive `prefix` that defines where your data will be uploaded in the default S3 bucket. Finally, create a pointer to your training data by calling `sagemaker_session.upload_data` and passing in the required parameters. It may help to look at the [Session documentation (https://sagemaker.readthedocs.io/en/stable/session.html#sagemaker.session.Session.upload_data)](https://sagemaker.readthedocs.io/en/stable/session.html#sagemaker.session.Session.upload_data) or previous SageMaker code examples.

You are expected to upload your entire directory. Later, the training script will only access the `train.csv` file.

In [4]:
```
# should be the name of directory you created to save your features data
data_dir = 'plagiarism_data'

# set prefix, a descriptive name for a directory
prefix = 'plagiarism-data-model'

# upload all data to S3
input_data = sagemaker_session.upload_data(path=data_dir, bucket=bucket, key_prefix=prefix)
```

Test cell

Test that your data has been successfully uploaded. The below cell prints out the items in your S3 bucket and will throw an error if it is empty. You should see the contents of your `data_dir` and perhaps some checkpoints. If you see any other files listed, then you may have some old model files that you can delete via the S3 console (though, additional files shouldn't affect the performance of model developed in this notebook).

In [5]:
```
"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
# confirm that data is in S3 bucket
empty_check = []
for obj in boto3.resource('s3').Bucket(bucket).objects.all():
    empty_check.append(obj.key)
    print(obj.key)

assert len(empty_check) !=0, 'S3 bucket is empty.'
print('Test passed!')
```

```
deepar-energy-consumption/output/forecasting-deepar-2019-11-27-09-34-57-460/output/model.tar.gz
deepar-energy-consumption/test/test.json
deepar-energy-consumption/train/train.json
plagiarism-data-model/test.csv
plagiarism-data-model/train.csv
Test passed!
```

Modeling

Now that you've uploaded your training data, it's time to define and train a model!

The type of model you create is up to you. For a binary classification task, you can choose to go one of three routes:

- Use a built-in classification algorithm, like LinearLearner.
- Define a custom Scikit-learn classifier, a comparison of models can be found [here (https://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html)](https://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html).
- Define a custom PyTorch neural network classifier.

It will be up to you to test out a variety of models and choose the best one. Your project will be graded on the accuracy of your final model.

EXERCISE: Complete a training script

To implement a custom classifier, you'll need to complete a `train.py` script. You've been given the folders `source_sklearn` and `source_pytorch` which hold starting code for a custom Scikit-learn model and a PyTorch model, respectively. Each directory has a `train.py` training script. To complete this project **you only need to complete one of these scripts**; the script that is responsible for training your final model.

A typical training script:

- Loads training data from a specified directory
- Parses any training & model hyperparameters (ex. nodes in a neural network, training epochs, etc.)
- Instantiates a model of your design, with any specified hyperparams
- Trains that model
- Finally, saves the model so that it can be hosted/deployed, later

Defining and training a model

Much of the training script code is provided for you. Almost all of your work will be done in the `if __name__ == '__main__':` section. To complete a `train.py` file, you will:

1. Import any extra libraries you need
2. Define any additional model training hyperparameters using `parser.add_argument`
3. Define a model in the `if __name__ == '__main__':` section
4. Train the model in that same section

Below, you can use `!pygmentize` to display an existing `train.py` file. Read through the code; all of your tasks are marked with `TODO` comments.

Note: If you choose to create a custom PyTorch model, you will be responsible for defining the model in the `model.py` file, and a `predict.py` file is provided. If you choose to use Scikit-learn, you only need a `train.py` file; you may import a classifier from the `sklearn` library.

In [6]:
```
# directory can be changed to: source_sklearn or source_pytorch
!pygmentize source_pytorch/train.py
```

```
import argparse
import json
import os
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
import torch.utils.data

# imports the model in model.py by name
from model import BinaryClassifier

def model_fn(model_dir):
    """Load the PyTorch model from the `model_dir` directory."""
    print("Loading model.")

    # First, load the parameters used to create the model.
    model_info = {}
    model_info_path = os.path.join(model_dir, 'model_info.pth')
```

Provided code

If you read the code above, you can see that the starter code includes a few things:

- Model loading (`model_fn`) and saving code
- Getting SageMaker's default hyperparameters
- Loading the training data by name, `train.csv` and extracting the features and labels, `train_x` , and `train_y`

If you'd like to read more about model saving with [joblib for sklearn (https://scikit-learn.org/stable/modules/model_persistence.html)](https://scikit-learn.org/stable/modules/model_persistence.html) or with [torch.save (https://pytorch.org/tutorials/beginner/saving_loading_models.html)](https://pytorch.org/tutorials/beginner/saving_loading_models.html), click on the provided links.

Create an Estimator

When a custom model is constructed in SageMaker, an entry point must be specified. This is the Python file which will be executed when the model is trained; the `train.py` function you specified above. To run a custom training script in SageMaker, construct an estimator, and fill in the appropriate constructor arguments:

- **entry_point**: The path to the Python script SageMaker runs for training and prediction.
- **source_dir**: The path to the training script directory `source_sklearn` OR `source_pytorch` .
- **entry_point**: The path to the Python script SageMaker runs for training and prediction.
- **source_dir**: The path to the training script directory `train_sklearn` OR `train_pytorch` .
- **entry_point**: The path to the Python script SageMaker runs for training.
- **source_dir**: The path to the training script directory `train_sklearn` OR `train_pytorch` .
- **role**: Role ARN, which was specified, above.
- **train_instance_count**: The number of training instances (should be left at 1).
- **train_instance_type**: The type of SageMaker instance for training. Note: Because Scikit-learn does not natively support GPU training, Sagemaker Scikit-learn does not currently support training on GPU instance types.
- **sagemaker_session**: The session used to train on Sagemaker.
- **hyperparameters** (optional): A dictionary `{'name':value, ..}` passed to the train function as hyperparameters.

Note: For a PyTorch model, there is another optional argument **framework_version**, which you can set to the latest version of PyTorch, `1.0` .

EXERCISE: Define a Scikit-learn or PyTorch estimator

To import your desired estimator, use one of the following lines:

```
from sagemaker.sklearn.estimator import SKLearn
```

```
from sagemaker.pytorch import PyTorch
```

In [14]:

```
# your import and estimator code, here
# import a PyTorch wrapper
from sagemaker.pytorch import PyTorch

# specify an output path
# prefix is specified above
output_path = 's3://{}/{}'.format(bucket, prefix)

estimator = PyTorch(entry_point='train.py',
                    source_dir='source_pytorch', # this should be just "source" for your code
                    role=role,
                    framework_version='1.0',
                    train_instance_count=1,
                    train_instance_type='ml.c4.xlarge',
                    output_path=output_path,
                    sagemaker_session=sagemaker_session,
                    hyperparameters={
                        'input_features': 3,  # num of features
                        'hidden_dim': 10,
                        'output_dim': 1,
                       # 'early_stopping_patience': 100,
                       # 'early_stopping_type':'Auto',
                        'epochs': 800 # could change to higher
                    })
```

EXERCISE: Train the estimator

Train your estimator on the training data stored in S3. This should create a training job that you can monitor in your SageMaker console.

In [15]:
```
%%time

# Train your estimator on S3 training data
estimator.fit({'train': input_data})
```

```
Epoch: 778, Loss: 0.3233901113271713

Epoch: 779, Loss: 0.3243270218372345
Epoch: 780, Loss: 0.37602536380290985
Epoch: 781, Loss: 0.3185868263244629
Epoch: 782, Loss: 0.3615611642599106
Epoch: 783, Loss: 0.34920623898506165
Epoch: 784, Loss: 0.37498287856578827
Epoch: 785, Loss: 0.4317897707223892
Epoch: 786, Loss: 0.3934497684240341
Epoch: 787, Loss: 0.41586238145828247
Epoch: 788, Loss: 0.3351696729660034
Epoch: 789, Loss: 0.36565282940864563
Epoch: 790, Loss: 0.28985877335071564
Epoch: 791, Loss: 0.3447311967611313
Epoch: 792, Loss: 0.32728809118270874
Epoch: 793, Loss: 0.34221889078617096
Epoch: 794, Loss: 0.36525481939315796
Epoch: 795, Loss: 0.2940495163202286
Epoch: 796, Loss: 0.43641915917396545
```

EXERCISE: Deploy the trained model

After training, deploy your model to create a `predictor` . If you're using a PyTorch model, you'll need to create a trained `PyTorchModel` that accepts the trained `<model>.model_data` as an input parameter and points to the provided `source_pytorch/predict.py` file as an entry point.

To deploy a trained model, you'll use `<model>.deploy` , which takes in two arguments:

- **initial_instance_count**: The number of deployed instances (1).
- **instance_type**: The type of SageMaker instance for deployment.

Note: If you run into an instance error, it may be because you chose the wrong training or deployment instance_type. It may help to refer to your previous exercise code to see which types of instances we used.

In [16]:

```
%%time

# uncomment, if needed
# from sagemaker.pytorch import PyTorchModel

from sagemaker.pytorch import PyTorchModel

# Create a model from the trained estimator data
# And point to the prediction script
model = PyTorchModel(model_data=estimator.model_data,
                     role = role,
                     framework_version='1.0',
                     entry_point='predict.py',
                     source_dir='source_pytorch')


# deploy your model to create a predictor
predictor = model.deploy(initial_instance_count=1, instance_type='ml.t2.medium')
```

```
--------------------------------------------------------------------------------------------------------------------------!CPU times: user 690 ms, sys: 65.1 ms, total: 755 ms
Wall time: 10min 22s
```

Evaluating Your Model

Once your model is deployed, you can see how it performs when applied to our test data.

The provided cell below, reads in the test data, assuming it is stored locally in `data_dir` and named `test.csv`. The labels and features are extracted from the `.csv` file.

In [17]:

```
"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
import os

# read in test data, assuming it is stored locally
test_data = pd.read_csv(os.path.join(data_dir, "test.csv"), header=None, names=None)

# labels are in the first column
test_y = test_data.iloc[:,0]
test_x = test_data.iloc[:,1:]
```

EXERCISE: Determine the accuracy of your model

Use your deployed `predictor` to generate predicted, class labels for the test data. Compare those to the *true* labels, `test_y`, and calculate the accuracy as a value between 0 and 1.0 that indicates the fraction of test data that your model classified correctly. You may use [sklearn.metrics (https://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics)](https://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics) for this calculation.

To pass this project, your model should get at least 90% test accuracy.

In [18]:
```
# First: generate predicted, class labels
import numpy as np
test_y_preds =np.squeeze(np.round(predictor.predict(test_x)))

    # calculate true positives, false positives, true negatives, false negatives
tp = np.logical_and(test_y, test_y_preds).sum()
fp = np.logical_and(1-test_y, test_y_preds).sum()
tn = np.logical_and(1-test_y, 1-test_y_preds).sum()
fn = np.logical_and(test_y, 1-test_y_preds).sum()

    # calculate binary classification metrics
recall = tp / (tp + fn)
precision = tp / (tp + fp)
accuracy = (tp + tn) / (tp + fp + tn + fn)

    # print metrics
print(pd.crosstab(test_y, test_y_preds, rownames=['actuals'], colnames=['predictions']))
print("\n{:<11} {:.3f}".format('Recall:', recall))
print("{:<11} {:.3f}".format('Precision:', precision))
print("{:<11} {:.3f}".format('Accuracy:', accuracy))
print()


"""
DON'T MODIFY ANYTHING IN THIS CELL THAT IS BELOW THIS LINE
"""
# test that your model generates the correct number of labels
assert len(test_y_preds)==len(test_y), 'Unexpected number of predictions.'
print('Test passed!')
```

```
predictions  0.0  1.0
actuals
0              9    1
1              0   15

Recall:     1.000
Precision:  0.938
Accuracy:   0.960

Test passed!
```

In [19]:
```
# Second: calculate the test accuracy
recall = tp / (tp + fn)
precision = tp / (tp + fp)
accuracy = (tp + tn) / (tp + fp + tn + fn)

print(accuracy)


## print out the array of predicted and true labels, if you want
print('\nPredicted class labels: ')
print(test_y_preds)
print('\nTrue class labels: ')
print(test_y.values)
```

```
0.96

Predicted class labels:
[1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 1. 1. 0. 1. 1. 1. 1. 0.
 0.]

True class labels:
[1 1 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 0 1 0 1 1 0 0]
```

Question 1: How many false positives and false negatives did your model produce, if any? And why do you think this is?

** Answer**: Recall: 1.000 Precision: 0.938 Accuracy: 0.960

although the accircy is improved in case of icreasing the number of epocks , bit i have some doupt about overfitting , that is why i am trying to make early stopping in hyperparameters tuning

Question 2: How did you decide on the type of model to use?

** Answer**: i decided to use Pytorch according to the good results in the previous projects, and i believe that it can be improved by some tunning for hyper parameters

---

EXERCISE: Clean up Resources

After you're done evaluating your model, **delete your model endpoint**. You can do this with a call to `.delete_endpoint()` . You need to show, in this notebook, that the endpoint was deleted. Any other resources, you may delete from the AWS console, and you will find more instructions on cleaning up all your resources, below.

In [20]:
```
# uncomment and fill in the line below!
# <name_of_deployed_predictor>.delete_endpoint()
def delete_endpoint(predictor):
        try:
            boto3.client('sagemaker').delete_endpoint(EndpointName=predictor.endpoint)
            print('Deleted {}'.format(predictor.endpoint))
        except:
            print('Already deleted: {}'.format(predictor.endpoint))
delete_endpoint(predictor)
```

```
Deleted sagemaker-pytorch-2019-11-27-16-37-47-378
```

Deleting S3 bucket

When you are *completely* done with training and testing models, you can also delete your entire S3 bucket. If you do this before you are done training your model, you'll have to recreate your S3 bucket and upload your training data again.

In [21]:
```
# deleting bucket, uncomment lines below

bucket_to_delete = boto3.resource('s3').Bucket(bucket)
bucket_to_delete.objects.all().delete()
```

Out[21]:

```
[{'ResponseMetadata': {'RequestId': '6FE0ED6342E7D761',
   'HostId': 'l32yUyOfku246ewRFxZ5UTXLXj1MCo28tKCsyfNKa5YFP1Z+n5WTEnwbQBBD2AMQu2QKOfB4jxc=',
   'HTTPStatusCode': 200,
   'HTTPHeaders': {'x-amz-id-2': 'l32yUyOfku246ewRFxZ5UTXLXj1MCo28tKCsyfNKa5YFP1Z+n5WTEnwbQBBD2AMQu2QKOfB4jxc=',
    'x-amz-request-id': '6FE0ED6342E7D761',
    'date': 'Wed, 27 Nov 2019 16:49:50 GMT',
    'connection': 'close',
    'content-type': 'application/xml',
    'transfer-encoding': 'chunked',
    'server': 'AmazonS3'},
   'RetryAttempts': 0},
  'Deleted': [{'Key': 'plagiarism-data-model/test.csv'},
   {'Key': 'plagiarism-data-model/train.csv'},
   {'Key': 'deepar-energy-consumption/test/test.json'},
   {'Key': 'sagemaker-pytorch-2019-11-27-16-33-10-611/source/sourcedir.tar.gz'},
   {'Key': 'deepar-energy-consumption/output/forecasting-deepar-2019-11-27-09-34-57-460/output/model.tar.gz'},
   {'Key': 'deepar-energy-consumption/train/train.json'},
   {'Key': 'sagemaker-pytorch-2019-11-27-16-28-59-733/source/sourcedir.tar.gz'},
   {'Key': 'sagemaker-pytorch-2019-11-27-16-24-53-148/source/sourcedir.tar.gz'},
   {'Key': 'plagiarism-data-model/sagemaker-pytorch-2019-11-27-16-33-10-611/output/model.tar.gz'},
   {'Key': 'sagemaker-pytorch-2019-11-27-16-37-46-871/sourcedir.tar.gz'}]}]
```


Deleting all your models and instances

When you are *completely* done with this project and do **not** ever want to revisit this notebook, you can choose to delete all of your SageMaker notebook instances and models by following [these instructions (https://docs.aws.amazon.com/sagemaker/latest/dg/ex1-cleanup.html)](https://docs.aws.amazon.com/sagemaker/latest/dg/ex1-cleanup.html). Before you delete this notebook instance, I recommend at least downloading a copy and saving it, locally.

Further Directions

There are many ways to improve or add on to this project to expand your learning or make this more of a unique project for you. A few ideas are listed below:

- Train a classifier to predict the *category* (1-3) of plagiarism and not just plagiarized (1) or not (0).
- Utilize a different and larger dataset to see if this model can be extended to other types of plagiarism.
- Use language or character-level analysis to find different (and more) similarity features.
- Write a complete pipeline function that accepts a source text and submitted text file, and classifies the submitted text as plagiarized or not.
- Use API Gateway and a lambda function to deploy your model to a web application.

These are all just options for extending your work. If you've completed all the exercises in this notebook, you've completed a real-world application, and can proceed to submit your project. Great job!