# CS985DLGL-GA2-MULT

April 11, 2021

# 1 Multi-Classification Problem

## 1.1 CS985 Deep Learning Group L

Ian Richardson 202074007

Fraser Bayne 202053049

Slav Ivanov 201645797

Lora Kiosseva 202082329

---

In short, our group implemented 5 models: - A standard ensamble method, Random Forest, - A baseline Neural Network - An LSTM-based model - A BERT-based model - A Word2Vec-based model

Our main finding is that the performance of the models is heavily based on the data augmentation. Models such as the BERT-based one performs very well, achieving Kaggle scores of ~0.92.

# 2 Method

The process we used for data processing includes a number of steps. After the .csv files are loaded, all labels with less than 1000 occurances are dropped. After examining the **label** column, we saw that the main labels(the ones that contain a single 1 in them, i.e **000000001**) were the ones that were predominantly represented.

Dropping everything below 1000 occurances fully covers the 9 main labels and, respectively, the 9 main categories of document. The original value at which the cutoff happened was 100. That left in 29 unique labels which reduced the trianing accuracy and loss score, as these other 20 labels were relatively under-represented and had less occurances to train on.

The next step involves dropping the columns named **'docid', 'publication_date', 'category', 'country_code', 'country_name', 'sector', 'value'.** From our inspection, these colums did not seem to add much to the data and we decided to remove them. Most of them are just unique identifiers within the dataframe, whereas others(such as country_name) simply had the same value inside. Others (value) had NaN values in them and it was not worth trying to fill them in.

Subsequently, the remaining columns are combined within a single String column called 'data'. **This way all the data for the particular label can be seen as a single string that can be converted to a single vector, or passed into BERT more easily.**

---

The dataset is split up into X and y, **where X is the 'data' column and y the 'label'.**

- The y dataset is then transformed by a LabelEncoder. This would enable having the final activation for the NNs to be **softmax**. This approach is very similar to the one used for the **Fashion-MNIST task described in the book Hands-On Machine Learning with Scikit-Learn, Keras and Tensorflow[1]**
- The X dataset is transformed into numeric vectors through a CountVectorizer. This data would be used in all models apart from the BERT and Word2Vec based ones.

This is then split up into testing and training sets through the use of train_test_split. The training set is split up into a **training and validation set**, to avoid overfitting.

---

## 2.1 EVERY NN USES THE SAME FINAL LAYER FOR CSV GENERATION AND TESTING CONSISTENCY

**The final layer contains neurons equal to however many unique labels were produced by the LabelEncoder(10 when the cutoff for labels is 1000 occurances, 29 when it is 100). The activation function is softmax.**

## 2.2 Baseline Model - function: rnd_for()

The baseline model of choice was a **scikit-learn ensamble method model - Random Forest**. It was chosen as it is considered to be one of the most versatile Machine Learning models. Despite it being simpler than a Neural Network(NN), it still manages to perform quite well in multi-class classification tasks, such as this one.

## 2.3 Baseline NN - class: BaselineNN()

The baseline NN of choice is a fully-connected, feedforward Neural Network, consisting of 3 layers of 300 neurons each. Each layer's activation function is ReLU. These 3 layers are followed by the output layer.

## 2.4 LSTM Model - class: LSTMModel()

The LSTM was chosen due to the fact that the data we were dealing with was in a text format. RNNs in general perform quite well at task like these and the LSTM being a better version of them was a natural choice. This LSTM model consists of 2 LSTM layers, followed by a Dense layer, that is ultimately followed by the output layer.

## 2.5 BERT Model - function: bertholomew()

This model utilises the fine-tuning of a BERT model[2]. The preprocessing unit and encoder are obtained through tfhub. The BERT model of choice is **bert_multi_cased_L-12_H-768_A-12/3** and its respective encoder. We chose it as it was case-sensitive, multilingual, and it had multiple large layers. The BERT layers are then followed by a combination of 3 Dense layers and Dropouts, ending in the final output layer.

### 2.6 Word2Vec - class: Word2Vec()

This model uses the text vectorization layer to normalize, split, and map strings from the dataset to integers. output_sequence_length length is used to pad all samples to same length. Sequence_length is set 10 as that's roughly how many words each row in the data has. If a row has less words, it is padded with 0s.

Once all the data has been integer encoded, skip-gram pairs with negative sampling for a list of sequences (int-encoded sentences) is generated based on window size, number of negative samples and vocabulary size as per [3] This gives us targets, contexts and labels, which word2vec is trained on.

Once word2vec is trained, it generates a matrix of wights (embedding matrix), which can be passed to an embedding layer of a deep learning model. The output of this Embedding layer is a 2D matrix with a word vector for each word id in the input sequence (Sequence length x Vector size). This is followed by two layers of Bidirectional LSTM to model the order of words in a sequence in both directions. Two final dense layers that predict the probability for each category. The model used is lrgely inspired by [4].

### 2.7 Predict - function: generate_csv

Finally, this function will take the trained model and get a prediction for the test dataset, writting it out to a CSV that can be submitted to Kaggle. From the values predicted we pick the one with the highest value predicted and get the index of the label at that position. Using this index, we can get the specific label from the list of unique labels and thus we have our end prediction.

---

### 2.8 Training - function: trainModel(model, xm_train, ym_train, xm_val, ym_val)

Training was done with 5 to 10 epochs of training for the models, due to the fact that some models take considerably longer than others to train. Sparse-categorical-crossentropy was used as the loss, as we had a single output neuron per output label. Accuracy was used as the metric and stochastic gradient descent as the optimiser. Adam was used at first but it converged slower and to a less desirable loss fucntion score.

Through experimentation with the models' hyperparameters, different values produced different results. The results gathered did not improve when changes were made, save from increasing training times.

---

### 2.9 Performance

Due to RAM limitations and crashed, variables are deleted and garbage collected when training of a model finishes. Due to this, the Random Forest had to be trained on only a 1000 row subset of the data.

[1]:
```
!pip install -q tensorflow-text
```

[2]:
```
!pip install -q tf-models-official
```

[1]:
```
import numpy as np
import pandas as pd
import gc

import tensorflow as tf
import tensorflow_hub as hub
import tensorflow_text as text
from tensorflow import keras

from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import f1_score, confusion_matrix

from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer

from sklearn.ensemble import RandomForestClassifier

tf.get_logger().setLevel('ERROR')
```

[2]:
```
data = pd.read_csv('./dataset/german-contracts-train.csv', dtype={
        "docid":str,
        "publication_date":str,
        "contract_type":str,
        "nature_of_contract":str,
        "country_code":str,
        "country_name":str,
        "sector":str,
        "category":str,
        "value":float,
        "title":str,
        "description":str,
        "awarding_authority":str,
        "complete_entry":str,
        "label":str})

test_data = pd.read_csv('./dataset/german-contracts-test.csv', dtype={
        "docid":str,
        "publication_date":str,
        "contract_type":str,
        "nature_of_contract":str,
        "country_code":str,
        "country_name":str,
        "sector":str,
```

```
        "category":str,
        "value":float,
        "title":str,
        "description":str,
        "awarding_authority":str,
        "complete_entry":str,
        "label":str})
id_column = np.array(test_data["docid"]).reshape(-1,1)
```

[3]:
```
data = data.groupby('label').filter(lambda x : len(x)>=1000)

# DROP COLUMNS
data = data.drop(columns=['docid', 'publication_date', 'category',␣
 ↪'country_code', 'country_name', 'sector', 'value'])
test_data = test_data.drop(columns=['docid', 'publication_date',␣
 ↪'country_code', 'country_name', 'sector', 'value'])

data = data.dropna()
test_data = test_data.fillna('nan')
```

[4]:
```
data['data'] = data['contract_type'] + ' ' + data['nature_of_contract'] + ' ' +␣
 ↪data['title'] + ' '
+ data['description'] + ' ' + data['awarding_authority']
data = data.drop(columns=['contract_type', 'nature_of_contract', 'title',␣
 ↪'description', 'awarding_authority'])
```

[5]:
```
test_data['data'] = test_data['contract_type'] + ' ' +␣
 ↪test_data['nature_of_contract'] + ' ' + test_data['title'] + ' '
+ test_data['description'] + ' ' + test_data['awarding_authority']
test_data = test_data.drop(columns=['contract_type', 'nature_of_contract',␣
 ↪'title', 'description', 'awarding_authority'])
```

[6]:
```
def vectorize_data(data, test_data):
  vectorizer = CountVectorizer(min_df=0, lowercase=False, analyzer='word')
  vectorizer.fit(data)
  return vectorizer.transform(data).toarray().astype(np.float32), vectorizer.
 ↪transform(test_data).toarray().astype(np.float32)
```

[7]:
```
def label_encode(labels):
  le = LabelEncoder()
  le.fit(labels)
  return np.unique(labels), le.transform(labels)
```

[8]:
```
# Get everything except what we want to predict
def prepare_baseline_data():
  X, X_forreal = vectorize_data(np.array(data['data']), np.
 ↪array(test_data['data']))
```

```python
    # Column we want to predict
    y_uniques, y = label_encode(np.array(data['label']))

    return X, X_forreal, y, y_uniques
```

[9]:
```python
def prepare_bert_data():
    # Get everything except what we want to predict
    X = np.array(data['data'])
    X_forreal = np.array(test_data['data'])
    # Column we want to predict
    y_uniques, y = label_encode(np.array(data['label']))

    return X, X_forreal, y, y_uniques
```

[10]:
```python
def rnd_for():
    cutoff = 1000
    model = RandomForestClassifier(random_state=42)
    model.fit(X_train_full[:cutoff, :], y_train_full[:cutoff])
    y_pred = model.predict(X_test[:cutoff, :])
    print('> Random Forest Classifier', model.score(X_test[:cutoff, :], y_test[:
↪cutoff]),
          '- F1', f1_score(y_test[:cutoff], y_pred, average='macro'))
    return model
```

[11]:
```python
class BaselineNN(keras.models.Model):
    def __init__(self, units=300, activation="relu", **kwargs):
        super().__init__(**kwargs) # handles standard args (e.g., name)
        self.hidden1 = keras.layers.Dense(units, activation=activation)
        self.hidden2 = keras.layers.Dense(units, activation=activation)
        self.hidden3 = keras.layers.Dense(units, activation=activation)
        self.main_output = keras.layers.Dense(len(y_uniques),␣
↪activation='softmax')

    def call(self, inputs):
        hidden1 = self.hidden1(inputs)
        hidden2 = self.hidden2(hidden1)
        hidden3 = self.hidden3(hidden2)
        main_output = self.main_output(hidden3)
        return main_output
```

[12]:
```python
 class LSTMModel(keras.models.Model):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.hidden1 = keras.layers.LSTM(256, return_sequences=True)
        self.hidden2 = keras.layers.Dropout(0.3)
        self.hidden3 = keras.layers.LSTM(128)
        self.hidden4 = keras.layers.Dense(256)
```

```
        self.hidden5 = keras.layers.Dropout(0.3)
        self.main_output = keras.layers.Dense(len(y_uniques),␣
↪activation='softmax')

    def call(self, inputs):
        hidden1 = self.hidden1(inputs)
        hidden2 = self.hidden2(hidden1)
        hidden3 = self.hidden3(hidden2)
        hidden4 = self.hidden4(hidden3)
        hidden5 = self.hidden5(hidden4)
        main_output = self.main_output(hidden5)
        return main_output
```

[13]:
```
def bertholomew():
    text_input = tf.keras.layers.Input(shape=(), dtype=tf.string, name='input')
    preprocessing_layer = hub.KerasLayer(tfhub_handle_preprocess,␣
↪name='preprocessing')
    encoder_inputs = preprocessing_layer(text_input)
    encoder = hub.KerasLayer(tfhub_handle_encoder, trainable=True,␣
↪name='BERT_encoder')
    outputs = encoder(encoder_inputs)
    net = outputs['pooled_output']
    net = tf.keras.layers.Dropout(0.1)(net)
    net = tf.keras.layers.Dense(300, activation="relu")(net)
    net = tf.keras.layers.Dropout(0.1)(net)
    net = tf.keras.layers.Dense(300, activation="relu")(net)
    net = tf.keras.layers.Dropout(0.1)(net)
    net = tf.keras.layers.Dense(300, activation="relu")(net)
    net = tf.keras.layers.Dense(len(y_uniques), activation='softmax',␣
↪name='classifier')(net)
    return tf.keras.Model(text_input, net)
```

[14]:
```
def trainModel(model, xm_train, ym_train, xm_val, ym_val):
    model.compile(loss='sparse_categorical_crossentropy', optimizer='sgd',␣
↪metrics=["accuracy"])
    b_size = 32
    eps = 5

    model.fit(xm_train, ym_train, batch_size=b_size, epochs=eps,␣
↪validation_data=(xm_val, ym_val))
    y_pred = model.predict(X_test)

    y_pred_classes = np.empty((0))
    y_pred_final = []
    y_test_final = []
    for element in y_pred:
```

```
    y_pred_classes = np.append(y_pred_classes, np.where(element == np.
↪amax(element))[0][0])

  for i in range(len(y_pred)):
    y_pred_final.append(y_uniques[y_pred_classes[i].astype(np.int64)])
    y_test_final.append(y_uniques[y_test[i].astype(np.int64)])

  y_pred_final = np.array(y_pred_final)
  y_test_final = np.array(y_test_final)
  print("F1 SCORE: ", f1_score(y_test_final, y_pred_final, average="macro"))
  return model
```

[15]:
```
def generate_csv(model, x, name):
  y_pred = model.predict(x)
  y_pred_final = []

  if(name == "sklearnrndfor"):
    for i in range(len(y_pred)):
      y_pred_final.append(y_uniques[y_pred[i].astype(np.int64)])
  else:
    y_pred_classes = np.empty((0))
    for element in y_pred:
      y_pred_classes = np.append(y_pred_classes, np.where(element == np.
↪amax(element))[0][0])

    for i in range(len(y_pred)):
      y_pred_final.append(y_uniques[y_pred_classes[i].astype(np.int64)])

  y_pred_final = np.array(y_pred_final)
  csv = np.concatenate((id_column, y_pred_final.reshape(-1,1)), axis=1)
  csv = np.vstack((np.array(["docid","label"]), csv))
  np.savetxt("./csv/" + name + ".csv", csv, fmt='%s', delimiter=",")
```

## 3 Results and Discussion

- **(Accuracy: 0.877 - f1: 0.69 - Kaggle: 0.80487)** - rnd_for()
- **(Accuracy: 0.949 - f1: 0.81 - Kaggle: 0.92204)** - BaselineNN()
- **(Accuracy: 0.599 - f1: 0.07 - Kaggle: 0.55385)** - LSTMModel()
- **(Accuracy: 0.978 - f1: 0.88 - Kaggle: 0.91853)** - bertholomew()
- **(Accuracy: 0.689 - f1: 0.20 - Kaggle: 0.60257)** - word2vec

---

In general, our findings show that the BaselineNN utilising CountVectorizer and BERT are very-much so neck and neck, in terms of Kaggle performance and general classification power.

Predicting labels directly through the use of a LabelEncoder is also another useful thing for us, as we do not have to combine the categories into the format that is requested. We simply get the index of the label that the Model is most confident is the predicted one(highest value amongst the output neurons). Using this index, we can get the specific label from the list of unique labels and thus we have our end prediction. **This approach is identical to the one seen in the Course AI book [1] for the Fashion-MNIST Data set(pages 294-295).**

The featurs selected to be most important to us are:

- contract_type
- nature_of_contract
- title
- description
- awarding_authority

Models tend to predominantly fail due to Memory concerns, not enabling them to trian, hance the deletion of variables. Low number of epochs, poorly configured optimizers(very low learning rates), and using substes of the data, expectedly, cause the models to underperform.

# 4 Summary and Recommendation

All in all, the system built can support more models than the ones described here. In terms of recommendations, the process of combining columns and vectorizing them proved to be a very robust one and it produces satisfactory results. The vectors produced can surely be used in different and more complex models. The combination of the data columns increased the scores that the BERT model produces as well. Last but not least, dropping any and all labels that were under-represented proved to be a succesfull strategy that increased accuracy scores across all models.

# 5 References


1. Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow, Aurélien Géron **(Idea for final layer consisting a neuron for each class, softmax activation function)**
2. https://www.tensorflow.org/tutorials/text/classify_text_with_bert - **BERT Model**
3. https://www.tensorflow.org/tutorials/text/word2vec - **Word2Vec**
4. https://towardsdatascience.com/text-classification-with-nlp-tf-idf-vs-word2vec-vs-bert-41ff868d1794 - **Word2Vec**


# 6 Training and Validating

## 6.1 Load in and split vectorized data

This data will be used in the following models: - Random Forest - BaselineNN - LSTM

[19]:
```
X, X_forreal, y, y_uniques = prepare_baseline_data()
X_train_full, X_test, y_train_full, y_test = train_test_split(X, y, test_size=0.
↪2, random_state=42)
X_valid, X_train = X_train_full[:4000], X_train_full[4000:]
y_valid, y_train = y_train_full[:4000], y_train_full[4000:]
del X, y
```

```
gc.collect()
```

[19]: 0

# 7 This the training of the baseline ML Model - rnd_for()

- This is run with only 1000 lines of training data

[20]:
```
modelSklearn = rnd_for()
generate_csv(modelSklearn, X_forreal, "sklearnrndfor")
del modelSklearn
gc.collect()
```

```
> Random Forest Classifier 0.877 - F1 0.6854916714523083
```

[20]: 108

# 8 This the training of the baseline NN Model - BaselineNN()

[21]:
```
modelBase = trainModel(BaselineNN(), X_train, y_train, X_valid, y_valid)
generate_csv(modelBase, X_forreal, "base")
del modelBase
gc.collect()
```

```
Epoch 1/5
2171/2171 [==============================] - 7s 3ms/step - loss: 1.2855 -
accuracy: 0.6258 - val_loss: 0.7182 - val_accuracy: 0.7897
Epoch 2/5
2171/2171 [==============================] - 6s 3ms/step - loss: 0.6405 -
accuracy: 0.8215 - val_loss: 0.4180 - val_accuracy: 0.8820
Epoch 3/5
2171/2171 [==============================] - 6s 3ms/step - loss: 0.3828 -
accuracy: 0.8937 - val_loss: 0.2907 - val_accuracy: 0.9202
Epoch 4/5
2171/2171 [==============================] - 6s 3ms/step - loss: 0.2594 -
accuracy: 0.9312 - val_loss: 0.2247 - val_accuracy: 0.9395
Epoch 5/5
2171/2171 [==============================] - 7s 3ms/step - loss: 0.2013 -
accuracy: 0.9489 - val_loss: 0.1858 - val_accuracy: 0.9492
F1 SCORE:  0.8111930078338488
```

[21]: 3267

# 9 This the training of the LSTM Model - LSTMModel()

- Data needs to first be reshaped to be used by this model

[22]:
```
lstm_in = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], 1))
lstm_val = np.reshape(X_valid, (X_valid.shape[0], X_valid.shape[1], 1))
X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))
X_forreal = np.reshape(X_forreal, (X_forreal.shape[0], X_forreal.shape[1], 1))
del X_train, X_valid
gc.collect()

modelLSTM = trainModel(LSTMModel(), lstm_in, y_train, lstm_val, y_valid)

generate_csv(modelLSTM, X_forreal, "lstm")
del X_forreal, y_uniques, X_test, y_test, modelLSTM
gc.collect()
```

```
Epoch 1/5
2171/2171 [==============================] - 1207s 554ms/step - loss: 1.5627 -
accuracy: 0.5912 - val_loss: 1.2295 - val_accuracy: 0.6055
Epoch 2/5
2171/2171 [==============================] - 1210s 558ms/step - loss: 1.5156 -
accuracy: 0.5947 - val_loss: 1.5055 - val_accuracy: 0.5985
Epoch 3/5
2171/2171 [==============================] - 1210s 558ms/step - loss: 1.5145 -
accuracy: 0.5964 - val_loss: 1.5036 - val_accuracy: 0.5985
Epoch 4/5
2171/2171 [==============================] - 1210s 557ms/step - loss: 1.5190 -
accuracy: 0.5932 - val_loss: 1.5034 - val_accuracy: 0.5985
Epoch 5/5
2171/2171 [==============================] - 1210s 557ms/step - loss: 1.5159 -
accuracy: 0.5941 - val_loss: 1.5046 - val_accuracy: 0.5985
F1 SCORE:  0.07414082576666095
```

[22]: 46989

# 10 This the training of the BERT-based Model - bertholomew()

- Data for BERT model is different from the ones used for the models above. BERT model works with strings directly, so new data needs to be loaded first.

[23]:
```
X, X_forreal, y, y_uniques = prepare_bert_data()
#del data, test_data #SEE IF THIS CAUSES ISSUE
#gc.collect()
X_train_full, X_test, y_train_full, y_test = train_test_split(X, y, test_size=0.
↪2, random_state=42)
X_valid, X_train = X_train_full[:4000], X_train_full[4000:]
y_valid, y_train = y_train_full[:4000], y_train_full[4000:]
del X, y
gc.collect()
```

[23]: 50

[24]:
```
tfhub_handle_encoder ='https://tfhub.dev/tensorflow/
↪bert_multi_cased_L-12_H-768_A-12/3'
tfhub_handle_preprocess = 'https://tfhub.dev/tensorflow/
↪bert_multi_cased_preprocess/3'
```

[25]:
```
modelBert = trainModel(bertholomew(), X_train, y_train, X_valid, y_valid)
generate_csv(modelBert, X_forreal, "bert")
del modelBert, X_train, y_train, X_valid,
y_valid, X_train_full, X_test, y_train_full, y_test,
X_forreal, y_uniques
gc.collect()
```

```
Epoch 1/5
2171/2171 [==============================] - 1949s 892ms/step - loss: 1.1032 -
accuracy: 0.6733 - val_loss: 0.2573 - val_accuracy: 0.9365
Epoch 2/5
2171/2171 [==============================] - 1934s 891ms/step - loss: 0.2417 -
accuracy: 0.9391 - val_loss: 0.1377 - val_accuracy: 0.9660
Epoch 3/5
2171/2171 [==============================] - 1933s 891ms/step - loss: 0.1439 -
accuracy: 0.9651 - val_loss: 0.1170 - val_accuracy: 0.9715
Epoch 4/5
2171/2171 [==============================] - 1935s 891ms/step - loss: 0.1119 -
accuracy: 0.9720 - val_loss: 0.1160 - val_accuracy: 0.9735
Epoch 5/5
2171/2171 [==============================] - 1910s 880ms/step - loss: 0.0940 -
accuracy: 0.9766 - val_loss: 0.0896 - val_accuracy: 0.9780
F1 SCORE:  0.8848392322463072
```

[25]: 75

## 11 Word2Vec based model

[26]:
```
import io
import re
import string
import tqdm

from tensorflow.keras import Model
from tensorflow.keras.layers import Dot, Embedding, Flatten
from tensorflow.keras.layers.experimental.preprocessing import TextVectorization
```

[27]:
```
def custom_standardization(input_data):
  lowercase = tf.strings.lower(input_data)
  return tf.strings.regex_replace(lowercase,
```

```
                              '[%s]' % re.escape(string.punctuation), '')
```

[28]:

```
vocab_size = 6626
embedding_dim = 100
num_ns = 4
```

[29]:

```
# Get everything except what we want to predict
X = np.array(data['data'])
X_forreal = np.array(test_data['data'])
# Column we want to predict
y_uniques, y = label_encode(np.array(data['label']))

sequence_length = 10

# Use the text vectorization layer to normalize, split, and map strings to
# integers. Set output_sequence_length length to pad all samples to same length.
vectorize_layer = TextVectorization(
    standardize=custom_standardization,
    max_tokens=vocab_size,
    output_mode='int',
    output_sequence_length=sequence_length)

text_ds = tf.data.Dataset.from_tensor_slices(X).filter(lambda x: tf.cast(tf.
 ↪strings.length(x), bool))

vectorize_layer.adapt(text_ds.batch(1024))
# Save the created vocabulary for reference.
inverse_vocab = vectorize_layer.get_vocabulary()

SEED = 42
AUTOTUNE = tf.data.AUTOTUNE
# Vectorize the data in text_ds.
text_vector_ds = text_ds.batch(1024).prefetch(AUTOTUNE).map(vectorize_layer).
 ↪unbatch()

X = list(text_vector_ds.as_numpy_iterator())
X = np.array(X)

X_train_full, X_test, y_train_full, y_test = train_test_split(X, y, test_size=0.
 ↪2, random_state=42)
X_valid, X_train = X_train_full[:4000], X_train_full[4000:]
y_valid, y_train = y_train_full[:4000], y_train_full[4000:]
```

[30]:

```
# Generates skip-gram pairs with negative sampling for a list of sequences
# (int-encoded sentences) based on window size, number of negative samples
# and vocabulary size.
def generate_training_data(sequences, window_size, num_ns, vocab_size, seed):
```

```
  # Elements of each training example are appended to these lists.
  targets, contexts, labels = [], [], []

  # Build the sampling table for vocab_size tokens.
  sampling_table = tf.keras.preprocessing.sequence.
↪make_sampling_table(vocab_size)

  # Iterate over all sequences (sentences) in dataset.
  for sequence in tqdm.tqdm(sequences):

    # Generate positive skip-gram pairs for a sequence (sentence).
    positive_skip_grams, _ = tf.keras.preprocessing.sequence.skipgrams(
          sequence,
          vocabulary_size=vocab_size,
          sampling_table=sampling_table,
          window_size=window_size,
          negative_samples=0)

    # Iterate over each positive skip-gram pair to produce training examples
    # with positive context word and negative samples.
    for target_word, context_word in positive_skip_grams:
      context_class = tf.expand_dims(
          tf.constant([context_word], dtype="int64"), 1)
      negative_sampling_candidates, _, _ = tf.random.
↪log_uniform_candidate_sampler(
          true_classes=context_class,
          num_true=1,
          num_sampled=num_ns,
          unique=True,
          range_max=vocab_size,
          seed=SEED,
          name="negative_sampling")

      # Build context and label vectors (for one target word)
      negative_sampling_candidates = tf.expand_dims(
          negative_sampling_candidates, 1)

      context = tf.concat([context_class, negative_sampling_candidates], 0)
      label = tf.constant([1] + [0]*num_ns, dtype="int64")

      # Append each element from the training example to global lists.
      targets.append(target_word)
      contexts.append(context)
      labels.append(label)

  return targets, contexts, labels
```

[31]:
```
text_ds = tf.data.Dataset.from_tensor_slices(X_forreal).filter(lambda x: tf.
↪cast(tf.strings.length(x), bool))
vectorize_layer.adapt(text_ds.batch(1024))

# Vectorize the data in text_ds.
text_vector_ds = text_ds.batch(1024).prefetch(AUTOTUNE).map(vectorize_layer).
↪unbatch()

X_forreal = list(text_vector_ds.as_numpy_iterator())

X_forreal = np.array(X_forreal)
```

[32]:
```
targets, contexts, labels = generate_training_data(
    sequences=X,
    window_size=2,
    num_ns=4,
    vocab_size=vocab_size,
    seed=SEED)

BATCH_SIZE = 1024
BUFFER_SIZE = 10000
dataset = tf.data.Dataset.from_tensor_slices(((targets, contexts), labels))
dataset = dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE, drop_remainder=True)

dataset = dataset.cache().prefetch(buffer_size=AUTOTUNE)
```

```
100%|      | 91826/91826 [01:09<00:00, 1324.81it/s]
```

[33]:
```
class Word2Vec(keras.Model):
  def __init__(self, vocab_size, embedding_dim):
    super(Word2Vec, self).__init__()
    self.target_embedding = keras.layers.Embedding(vocab_size,
                                      embedding_dim,
                                      input_length=1,
                                      name="w2v_embedding")
    self.context_embedding = keras.layers.Embedding(vocab_size,
                                       embedding_dim,
                                       input_length=num_ns+1)
    self.dots = keras.layers.Dot(axes=(3, 2))
    self.flatten = keras.layers.Flatten()

  def call(self, pair):
    target, context = pair
    we = self.target_embedding(target)
    ce = self.context_embedding(context)
    dots = self.dots([ce, we])
    return self.flatten(dots)
```

[34]:
```
def custom_loss(x_logit, y_true):
      return tf.nn.sigmoid_cross_entropy_with_logits(logits=x_logit,␣
 ↪labels=y_true)
```

[35]:
```
embedding_dim = 128
word2vec = Word2Vec(vocab_size, embedding_dim)
word2vec.compile(optimizer='adam',
                 loss=tf.keras.losses.CategoricalCrossentropy(from_logits=True),
                 metrics=['accuracy'])

tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir="logs")

word2vec.fit(dataset, epochs=10, callbacks=[tensorboard_callback])

weights = word2vec.get_layer('w2v_embedding').get_weights()[0]
```

```
Epoch 1/10
125/125 [==============================] - 3s 22ms/step - loss: 1.5881 -
accuracy: 0.4238
Epoch 2/10
125/125 [==============================] - 2s 15ms/step - loss: 1.2699 -
accuracy: 0.6838
Epoch 3/10
125/125 [==============================] - 2s 15ms/step - loss: 0.9539 -
accuracy: 0.7214
Epoch 4/10
125/125 [==============================] - 2s 15ms/step - loss: 0.7839 -
accuracy: 0.7641
Epoch 5/10
125/125 [==============================] - 2s 15ms/step - loss: 0.6726 -
accuracy: 0.7940
Epoch 6/10
125/125 [==============================] - 2s 16ms/step - loss: 0.5922 -
accuracy: 0.8166
Epoch 7/10
125/125 [==============================] - 2s 15ms/step - loss: 0.5309 -
accuracy: 0.8342
Epoch 8/10
125/125 [==============================] - 2s 15ms/step - loss: 0.4825 -
accuracy: 0.8478
Epoch 9/10
125/125 [==============================] - 2s 15ms/step - loss: 0.4432 -
accuracy: 0.8589
Epoch 10/10
125/125 [==============================] - 2s 15ms/step - loss: 0.4105 -
accuracy: 0.8690
```

[37]:
```python
training = model.fit(x = X_train, y=y_train, batch_size=256,
                     epochs=10, shuffle=True, verbose=1,
                     validation_data=(X_valid, y_valid))

y_pred = model.predict(X_test)
y_pred_classes = np.empty((0))
y_pred_final = []
y_test_final = []
for element in y_pred:
  y_pred_classes = np.append(y_pred_classes, np.where(element == np.
 ↪amax(element))[0][0])

for i in range(len(y_pred)):
  y_pred_final.append(y_uniques[y_pred_classes[i].astype(np.int64)])
  y_test_final.append(y_uniques[y_test[i].astype(np.int64)])

y_pred_final = np.array(y_pred_final)
y_test_final = np.array(y_test_final)
print("F1 SCORE: ", f1_score(y_test_final, y_pred_final, average="macro"))
```

```
Epoch 1/10
272/272 [==============================] - 9s 14ms/step - loss: 1.9494 -
accuracy: 0.5484 - val_loss: 1.4719 - val_accuracy: 0.5985
Epoch 2/10
272/272 [==============================] - 3s 10ms/step - loss: 1.4433 -
accuracy: 0.5954 - val_loss: 1.2715 - val_accuracy: 0.5985
Epoch 3/10
272/272 [==============================] - 3s 10ms/step - loss: 1.2408 -
accuracy: 0.5915 - val_loss: 1.1049 - val_accuracy: 0.5985
Epoch 4/10
272/272 [==============================] - 3s 10ms/step - loss: 1.1017 -
accuracy: 0.5986 - val_loss: 1.0462 - val_accuracy: 0.6298
Epoch 5/10
272/272 [==============================] - 3s 10ms/step - loss: 1.0526 -
accuracy: 0.6311 - val_loss: 1.0141 - val_accuracy: 0.6398
Epoch 6/10
272/272 [==============================] - 3s 10ms/step - loss: 1.0263 -
accuracy: 0.6354 - val_loss: 0.9918 - val_accuracy: 0.6503
Epoch 7/10
272/272 [==============================] - 3s 10ms/step - loss: 1.0071 -
accuracy: 0.6490 - val_loss: 0.9748 - val_accuracy: 0.6720
Epoch 8/10
272/272 [==============================] - 3s 10ms/step - loss: 0.9867 -
accuracy: 0.6717 - val_loss: 0.9607 - val_accuracy: 0.6860
Epoch 9/10
272/272 [==============================] - 3s 10ms/step - loss: 0.9822 -
accuracy: 0.6829 - val_loss: 0.9470 - val_accuracy: 0.6980
Epoch 10/10
```

```
272/272 [==============================] - 3s 10ms/step - loss: 0.9664 -
accuracy: 0.6897 - val_loss: 0.9333 - val_accuracy: 0.6980
```

[38]:
```
generate_csv(model, X_forreal, "word2vec")
```