Business Understainding

As more AI-powered tools that can produce essays, reports, and other academic content become available, students might be employing these technologies to do their assignments. While in some situations AI can improve learning, when these tools are abused for academic dishonesty, the educational process is compromised. Schools must recognize assignments that were created by AI instead of by students in order to protect academic integrity.

Problem statement

The school's goal is to create a system that can identify assignments that were probably created using artificial intelligence (AI) tools automatically. The objective is to guarantee that students turn in their own work in order to uphold academic standards.

Objectives

- **Get the model with high recall**
- **Get a model with high precision**
- **High accuracy**

Success criteria

- **Precision -> above 90%**
- **Recall -> above 90%**
- **Accuracy -> above 90%**

Data Understanding

The data consist of four datasets with text of varying word count.

- **Part1 -> 20 - 100 Words**
- **Part2 -> 100 - 200 Words**
- **Part3 -> 200 - 300 Words**
- **Part4 -> 300+ Words**

The first column of CSV contains the text and the second holds its respective class. The class column contains a binary value. Here, `0 denotes human-written` and `1 AI-generated text`.

Source of the data is [Github]

In [1]:

```
import pandas as pd

df_1 = pd.read_csv('Datasets/DatasetPart1.csv')
df_2 = pd.read_csv('Datasets/DatasetPart2.csv')
df_3 = pd.read_csv('Datasets/DatasetPart3.csv')
df_4 = pd.read_csv('Datasets/DatasetPart4.csv')

# merging the dfs

df = pd.concat([df_1, df_2, df_3, df_4])

df
```

Out[1]:

	Text	Class

	Text	Class
0	Mississippi Highway 403 and MS 15 in Mathiston...	0
1	State Route 97 northwest of Congress northeast...	0
2	Edwin B. Erickson III was an American politici...	0
3	The Tiger Fire was a wildfire that burned 16,2...	0
4	Fajsz , was Grand Prince of the Hungarians fro...	0
...	...	...
9995	Women in early modern Scotland, between the Re...	0
9996	The Court of Common Pleas was one of the four ...	1
9997	Megara is a fictional character from the Disne...	1
9998	Luan Da is a Chinese literary figure who lived...	1
9999	The New York Herald Tribune was a newspaper pu...	0

44138 rows × 2 columns

EDA

In [2]:

```
from nltk import FreqDist
from nltk import word_tokenize
```

In [3]:

```
# separate the two classes to compare the texts
human_df = df.loc[df['Class'] == 0]
ai_df = df.loc[df.Class == 1]
```

In [4]:

```
# tokenize the text
human_df['tokens'] = human_df.Text.apply(word_tokenize)
ai_df['tokens'] = ai_df.Text.apply(word_tokenize)

human_df.head()
```

```
C:\Users\mutis\AppData\Local\Temp\ipykernel_26004\2212118046.py:2: SettingWithCopyWarning
:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_g
uide/indexing.html#returning-a-view-versus-a-copy
  human_df['tokens'] = human_df.Text.apply(word_tokenize)
C:\Users\mutis\AppData\Local\Temp\ipykernel_26004\2212118046.py:3: SettingWithCopyWarning
:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_g
uide/indexing.html#returning-a-view-versus-a-copy
  ai_df['tokens'] = ai_df.Text.apply(word_tokenize)
```

Out[4]:

	Text	Class	tokens
0	Mississippi Highway 403 and MS 15 in Mathiston...	0	[Mississippi, Highway, 403, and, MS, 15, in, M...
1	State Route 97 northwest of Congress northeast...	0	[State, Route, 97, northwest, of, Congress, no...
2	Edwin B. Erickson III was an American politici...	0	[Edwin, B., Erickson, III, was, an, American, ...
3	The Tiger Fire was a wildfire that burned 16,2...	0	[The, Tiger, Fire, was, a, wildfire, that, bur...
4	Fajsz , was Grand Prince of the Hungarians fro...	0	[Fajsz, ,, was, Grand, Prince, of, the, Hungar...

Words distribution

In [5]:

```
# counting the words in both dfs
human_words_count = FreqDist(human_df.tokens.explode())
ai_words_count = FreqDist(ai_df.tokens.explode())
```

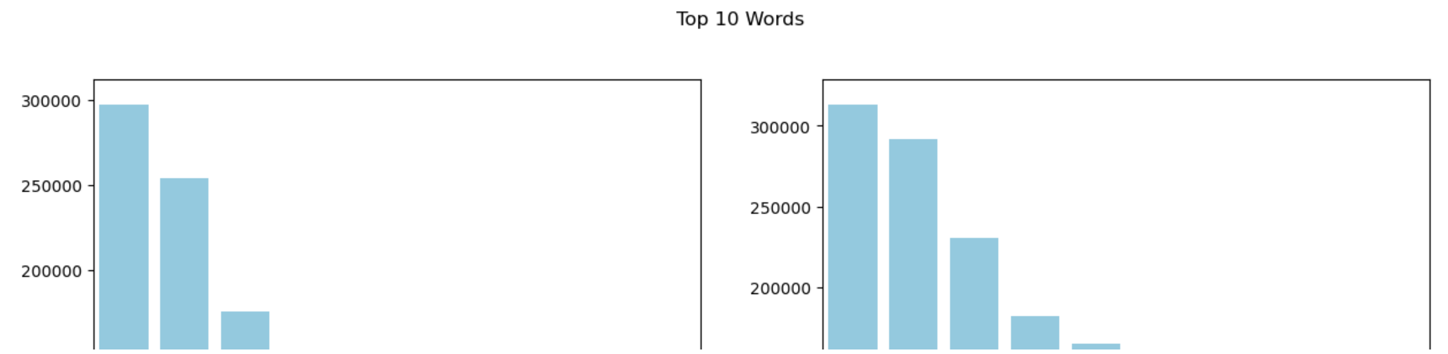
In [6]:

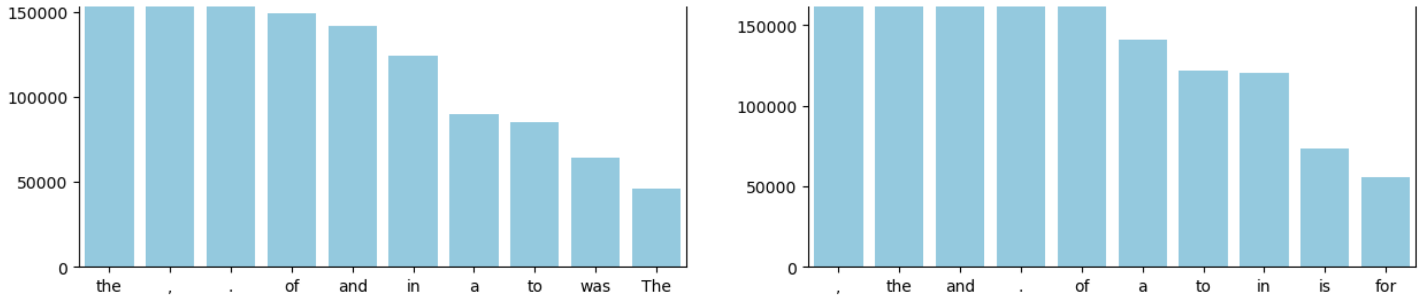
```
# plotting top 10 words for each class
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

def plot_most_common(human_count, ai_count, n=10):
    """Plots the disribution of the most common words

    Args:
        human_count: FreqDist object
        ai_count: FreqDist object
    """
    fig, ax = plt.subplots(figsize=(15, 6), ncols=2, nrows=1)

    top_10_human = human_count.most_common(n)
    top_10_ai = ai_count.most_common(n)

    # human text
    sns.barplot(
        x=[obj[0] for obj in top_10_human],
        y=[obj[1] for obj in top_10_human],
        color='skyblue',
        ax=ax[0]
    )

    # ai text
    sns.barplot(
        x=[obj[0] for obj in top_10_ai],
        y=[obj[1] for obj in top_10_ai],
        color='skyblue',
        ax=ax[1]
    )

    plt.suptitle(f'Top {n} Words');
```

In [7]:

```
plot_most_common(human_words_count, ai_words_count)
```

```
c:\Users\mutis\anaconda3\Lib\site-packages\seaborn\_oldcore.py:1765: FutureWarning: unique with argument that is not not a Series, Index, ExtensionArray, or np.ndarray is deprecated and will raise in a future version.
  order = pd.unique(vector)
c:\Users\mutis\anaconda3\Lib\site-packages\seaborn\_oldcore.py:1765: FutureWarning: unique with argument that is not not a Series, Index, ExtensionArray, or np.ndarray is deprecated and will raise in a future version.
  order = pd.unique(vector)
```





Notice that most common words in both sets are similar stopwords and punctuation.

In [8]:

```
from sklearn.model_selection import train_test_split
X = df[['Text']]
y = df.Class

# splitting the data
X_train, X_test, y_train, y_test = train_test_split(X, y)
```

Baseline model with original data

In [9]:

```
# vectorization
from sklearn.naive_bayes import MultinomialNB
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import cross_val_score

vectorizer = TfidfVectorizer()
vectorized_base_X = vectorizer.fit_transform(X_train['Text'])
```

In [10]:

```
# fitting the model
base_model = MultinomialNB()

base_model.fit(vectorized_base_X, y_train)
```

Out[10]:

▾ MultinomialNB i ?

```
MultinomialNB()
```

In [11]:

```
# validation
baseline_score = cross_val_score(base_model,
                                 vectorized_base_X,
                                 y_train, cv=2)
print('Baseline model score:', baseline_score.mean())
```

```
Baseline model score: 0.8828503471657343
```

Baseline model records a good score, about 88% accuracy.

a. Removing Stopwords and punctuation

In [12]:

```
from nltk.corpus import stopwords
import string
```

```
# only english
stopwords = stopwords.words('english')

# adding punctuation

stopwords += list(string.punctuation)
stopwords[-1]
```

Out[12]:

```
'~'
```

In [13]:

```
# remove stopwords
no_stopwords_vectiorizer = TfidfVectorizer(stop_words=stopwords)

no_stopwords_train_X = no_stopwords_vectiorizer.fit_transform(X_train['Text'])

# model
stopwords_model = MultinomialNB()

# crossvalidation
cross_val_score(stopwords_model,
                no_stopwords_train_X,
                y_train, cv=2).mean()
```

Out[13]:

```
0.8869285155314275
```

The model accuracy increased slightly after removing stopwords.

b. Stemming and lemmatization

In [14]:

```
import nltk
from nltk.stem import PorterStemmer
nltk.download('wordnet')
from nltk.stem.wordnet import WordNetLemmatizer
from nltk import word_tokenize

stemmer = PorterStemmer()
wnl = WordNetLemmatizer()

X_train_b = X_train.copy()

# function to stem and lemmatize words
def stem_lemmatize(text):
    tokens = word_tokenize(text)
    # stemming
    tokens = [stemmer.stem(x) for x in tokens]
    # lemmatize
    tokens = [wnl.lemmatize(x) for x in tokens]

    # join back to text
    new_text = " ".join(tokens)

    return new_text
```

```
[nltk_data] Downloading package wordnet to
[nltk_data]     C:\Users\mutis\AppData\Roaming\nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
```

In [15]:

```
# stem and lemmatize X train
X_train_b['Text'] = X_train_b.Text.apply(stem_lemmatize)
```

In [16]:

```
b_model = MultinomialNB()

# vectorization
b_vectorized_X = no_stopwords_vectiorizer.fit_transform(X_train_b['Text'])
```

In [17]:

```
# validation
cross_val_score(b_model, b_vectorized_X,
                y_train, cv=2).mean()
```

Out[17]:

```
0.8698606152946008
```

Stemming and lemmatization leads to lower accuracy.

c. Adding POS tags

This can add more info eg(when word is used as a noun or a verb)

In [18]:

```
from nltk import pos_tag
X_train_c = X_train.copy()

# function to add POS tags
def add_pos(text):
    """Adds POS tags to the text
    Args:
        text: str
    Returns:
        str: text with POS tags
    """
    # tokenization
    tokens = word_tokenize(text)
    # obtain tags
    pos_tags = pos_tag(tokens)
    tags = [x[1] for x in pos_tags]

    # add tags
    pos_added_tokens = ['_'.join(item) for item in list(zip(tokens, tags))]

    # join to single string
    pos_text = ' '.join(pos_added_tokens)

    return pos_text
```

In [19]:

```
# applying function to X_train
X_train_c['processed_text'] = X_train.Text.apply(add_pos)
X_train_c.head()
```

Out[19]:

	Text	processed_text
1089	"Dead Man Walking" is the fifth episode of the...	``_`` Dead_JJ Man_NN Walking_VBG ''_'' is_VBZ ...
6445	In neuroscience, long-term potentiation is a p...	In_IN neuroscience_NN ,_, long-term_JJ potenti...
15623	The 1893 New York hurricane, also known as the...	The_DT 1893_CD New_NNP York_NNP hurricane_NN ,...
6192	The 2001 Malta Grand Prix was a highly anticip...	The_DT 2001_CD Malta_NNP Grand_NNP Prix_NNP wa...
4459	HD 217107 c is an extrasolar planet approximat...	HD_$ 217107_CD c_NN is_VBZ an_DT extrasolar_JJ...

In [20]:

```
c_model = MultinomialNB()

final_vectorizer = TfidfVectorizer(stop_words=stopwords)
# vectorization
c_processed_X = final_vectorizer.fit_transform(X_train_c['processed_text'])
```

In [21]:

```
# validation
cross_val_score(c_model,
                c_processed_X,
                y_train,
                cv=2).mean()
```

Out[21]:

```
0.8860525012411656
```

We can see some improvement after adding the tags.

From all the preprocessing steps the effective ones are:

1. `Removing stopwords and punctuation.`
2. `Adding POS tags`

In [22]:

```
final_X_train = c_processed_X
```

In [23]:

```
# processing test data
# remove stopwards, punctuation and adding pos tags
X_test['processed_text'] = X_test.Text.apply(add_pos)

# vectorizing
X_test_preprocessed = final_vectorizer.transform(X_test['processed_text'])
```

Fitting other models

1. Logistic Regression

In [24]:

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import (
    accuracy_score,
    recall_score,
    precision_score
)
# model
logreg = LogisticRegression(solver='liblinear')

# fitting
logreg.fit(final_X_train, y_train)

preds = logreg.predict(X_test_preprocessed)

# evaluation
print('Logistic Regression Accuracy:', accuracy_score(y_test, preds))
print('Logistic Regression Recall:', recall_score(y_test, preds))
print('Logistic Regression Precision:', precision_score(y_test, preds))
```

```
Logistic Regression Accuracy: 0.9652016311735387
Logistic Regression Recall: 0.9560499186109603
```

```
Logistic Regression Recall: 0.9560499180109605
Logistic Regression Precision: 0.9740187949143173
```

2. Random Forest

In [25]:

```python
from sklearn.ensemble import RandomForestClassifier

# model
rf = RandomForestClassifier()

# fitting
rf.fit(final_X_train, y_train)

preds_rf = rf.predict(X_test_preprocessed)

# evaluation
print('Random Forest Accuracy:', accuracy_score(y_test, preds_rf))
print('Random Forest Recall:', recall_score(y_test, preds_rf))
print('Random Forest Precision:', precision_score(y_test, preds_rf))
```

```
Random Forest Accuracy: 0.953330312641595
Random Forest Recall: 0.9522517634291915
Random Forest Precision: 0.9544960116026106
```

3. XGBoost

In [26]:

```python
# !pip install xgboost
```

In [27]:

```python
from xgboost import XGBClassifier


# model
xgb = XGBClassifier()

# fitting
xgb.fit(final_X_train, y_train)

preds_xgb = xgb.predict(X_test_preprocessed)

# evaluation
print('XGBoost Accuracy:', accuracy_score(y_test, preds_xgb))
print('XGBoost Recall:', recall_score(y_test, preds_xgb))
print('XGBoost Precision:', precision_score(y_test, preds_xgb))
```

```
XGBoost Accuracy: 0.9570457634798368
XGBoost Recall: 0.9489962018448183
XGBoost Precision: 0.9646993932708219
```

4. Logistic regression as a neural network

In [28]:

```python
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense


# Build the Logistic Regression model using Keras
model = Sequential()
 # Logistic regression with 1 neuron and sigmoid activation
```

```
model.add(Dense(1, activation='sigmoid'))

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Train the model
model.fit(final_X_train, y_train, epochs=30, batch_size=10)

# Make predictions
predictions = model.predict(X_test_preprocessed)
# Apply 0.5 threshold
predicted_classes = np.where(predictions > 0.5, 1, 0)


# evaluation
print('Keras Accuracy:', accuracy_score(y_test, predicted_classes))
print('Keras Recall:', recall_score(y_test, predicted_classes))
print('Keras Precision:', precision_score(y_test, predicted_classes))
```

```
Epoch 1/30
3311/3311 ━━━━━━━━━━━━━━━━━━━━ 14s 4ms/step - accuracy: 0.8830 - loss: 0.5838
Epoch 2/30
3311/3311 ━━━━━━━━━━━━━━━━━━━━ 12s 4ms/step - accuracy: 0.9571 - loss: 0.3203
Epoch 3/30
3311/3311 ━━━━━━━━━━━━━━━━━━━━ 12s 4ms/step - accuracy: 0.9732 - loss: 0.2130
Epoch 4/30
3311/3311 ━━━━━━━━━━━━━━━━━━━━ 14s 4ms/step - accuracy: 0.9788 - loss: 0.1568
Epoch 5/30
3311/3311 ━━━━━━━━━━━━━━━━━━━━ 12s 4ms/step - accuracy: 0.9838 - loss: 0.1221
Epoch 6/30
3311/3311 ━━━━━━━━━━━━━━━━━━━━ 13s 4ms/step - accuracy: 0.9876 - loss: 0.0985
Epoch 7/30
3311/3311 ━━━━━━━━━━━━━━━━━━━━ 13s 4ms/step - accuracy: 0.9906 - loss: 0.0818
Epoch 8/30
3311/3311 ━━━━━━━━━━━━━━━━━━━━ 14s 4ms/step - accuracy: 0.9916 - loss: 0.0695
Epoch 9/30
3311/3311 ━━━━━━━━━━━━━━━━━━━━ 14s 4ms/step - accuracy: 0.9922 - loss: 0.0609
Epoch 10/30
3311/3311 ━━━━━━━━━━━━━━━━━━━━ 13s 4ms/step - accuracy: 0.9936 - loss: 0.0520
Epoch 11/30
3311/3311 ━━━━━━━━━━━━━━━━━━━━ 13s 4ms/step - accuracy: 0.9944 - loss: 0.0460
Epoch 12/30
3311/3311 ━━━━━━━━━━━━━━━━━━━━ 13s 4ms/step - accuracy: 0.9956 - loss: 0.0400
Epoch 13/30
3311/3311 ━━━━━━━━━━━━━━━━━━━━ 14s 4ms/step - accuracy: 0.9966 - loss: 0.0360
Epoch 14/30
3311/3311 ━━━━━━━━━━━━━━━━━━━━ 13s 4ms/step - accuracy: 0.9971 - loss: 0.0323
Epoch 15/30
3311/3311 ━━━━━━━━━━━━━━━━━━━━ 13s 4ms/step - accuracy: 0.9973 - loss: 0.0298
Epoch 16/30
3311/3311 ━━━━━━━━━━━━━━━━━━━━ 13s 4ms/step - accuracy: 0.9976 - loss: 0.0259
Epoch 17/30
3311/3311 ━━━━━━━━━━━━━━━━━━━━ 13s 4ms/step - accuracy: 0.9982 - loss: 0.0240
Epoch 18/30
3311/3311 ━━━━━━━━━━━━━━━━━━━━ 13s 4ms/step - accuracy: 0.9985 - loss: 0.0214
Epoch 19/30
3311/3311 ━━━━━━━━━━━━━━━━━━━━ 13s 4ms/step - accuracy: 0.9990 - loss: 0.0192
Epoch 20/30
3311/3311 ━━━━━━━━━━━━━━━━━━━━ 13s 4ms/step - accuracy: 0.9991 - loss: 0.0173
Epoch 21/30
3311/3311 ━━━━━━━━━━━━━━━━━━━━ 14s 4ms/step - accuracy: 0.9991 - loss: 0.0158
Epoch 22/30
3311/3311 ━━━━━━━━━━━━━━━━━━━━ 13s 4ms/step - accuracy: 0.9994 - loss: 0.0142
Epoch 23/30
3311/3311 ━━━━━━━━━━━━━━━━━━━━ 13s 4ms/step - accuracy: 0.9993 - loss: 0.0131
Epoch 24/30
3311/3311 ━━━━━━━━━━━━━━━━━━━━ 13s 4ms/step - accuracy: 0.9994 - loss: 0.0122
Epoch 25/30
3311/3311 ━━━━━━━━━━━━━━━━━━━━ 13s 4ms/step - accuracy: 0.9992 - loss: 0.0115
Epoch 26/30
3311/3311 ━━━━━━━━━━━━━━━━━━━━ 13s 4ms/step - accuracy: 0.9994 - loss: 0.0104
Epoch 27/30
```

```
3311/3311 ━━━━━━━━━━━━━━━━━━━━ 14s 4ms/step - accuracy: 0.9997 - loss: 0.0091
Epoch 28/30
3311/3311 ━━━━━━━━━━━━━━━━━━━━ 13s 4ms/step - accuracy: 0.9997 - loss: 0.0085
Epoch 29/30
3311/3311 ━━━━━━━━━━━━━━━━━━━━ 13s 4ms/step - accuracy: 0.9996 - loss: 0.0078
Epoch 30/30
3311/3311 ━━━━━━━━━━━━━━━━━━━━ 13s 4ms/step - accuracy: 0.9998 - loss: 0.0070
345/345 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step
Keras Accuracy: 0.9793384685092886
Keras Recall: 0.9737746427925483
Keras Precision: 0.984817998902506
```

This is the model with the best performance.

Saving model and preprocessing steps for deployment

In [29]:

```python
# save the model
model.save('model.h5')
```

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
```

In [30]:

```python
# save the vectorizer
import joblib
with open('vectorizer.pkl', 'wb') as file:
    joblib.dump(final_vectorizer, file)
```

In [31]:

```python
model.predict(final_vectorizer.transform([add_pos('I am a human')]))[0][0]
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 42ms/step
```

Out[31]:

```
0.019988097
```

In [ ]:

```python

```