Overview

This classification project aims to train several neural network architectures and machine learning models to determine the relevance of news articles to given search queries. For this, we train models on a dataset of web-scraped online news articles and their associated searches and relevance score.

To ensure better performance from our models, data cleaning is first carried out to remove irrelevant features, such as styling code. Using word vectorisation, we first explore standard machine learning techniques such as random forest decision trees and support vector machines. Then, using a combination of article and query information, we test three-layer neural networks to gain an enhanced appreciation for the dimensionality of the dataset and then a deep neural network. Upon separating the input data, we add more layers with wide and deep neural networks. Finally, we test recurrent complex models, namely LSTM and GRU, to exploit retained patterns (or lack thereof) in the data to influence our results.

We find that the relu activation function consistently performs well to ensure learning gradients of parameters do not become increasingly small. Equally, considering all our neural network models, we find the optimal number of neurons (computational units) to consistently be between 100 and 256. Overall, the standard baseline random forest model marginally performed the greatest. One of the recommendations made to improve the performance of the model is to invest resources into pre-trained models.

Method

Neural networks offer the advantage of sequential training with various dataset sections. Hence, we split the article information and corresponding query information into separate sections. We also kept aside its relevancy metric too. This was done because article bodies for given topics varied within their group and could concern different contexts. Equally, whilst initial inspections showed that the relevance score was typically accurate, there were outlier cases where this score was incorrect.

The bulk of the information in this dataset is text-based except for 'judgement' (binary relevancy scores) and 'topic_id' (nominal categories for queries). The 'body' column required the most cleaning because it included HTML and CSS. Removal of these prevented irrelevant and constant features from affecting the models's accuracies and training performance.

The dataset was made lowercase to ensure uniform capitalisation amongst terms with identical meanings. Using BeautifulSoup's html parser, HTML was removed. As CSS is occasionally retained within an embedded stylesheet, observations were split into lists at line breaks. This ensured CSS was contained then removed within lists starting with full stops, tab whitespaces, ampersands, etc., whilst legitimate text data lists started with alphanumeric characters. The next step was to remove punctuation and normalise Unicode hex values with the NFKD algorithm.

Common English words (called 'stopwords') were removed to ensure that unimportant words were not included lest our later models generalise poorly and train ineffectively. To reduce data dimensionality and ensure consistent meaning amongst words is not lost, this project uses Natural Language Toolkit libraries to lemmatise our data.

TfIDf Vectorisation was applied to the cleaned features. The TfIDf Vectorised features were used only in the standard machine learning baseline as this method of vectorisation produced too many inputs (Azzopardi, 2024) which was not efficient to run neutral networks. Hence, Word2Vec was used to vectorise the features for the remainder of the models. For both methods of vectorisation, the combined columns for queries and articles were tokenised separately for the train and test set.

From the exploratory data analysis, it was found that 84% of the judgement features were not relevant and 16% were relevant. Therefore, stratified validation was adopted as it addresses bias in the target features during the evaluation of the model (Géron, 2019).

Packages

In [ ]:

```
pip install pyarrow
```

In [ ]:
```
pip install wordcloud
```

In [ ]:
```
# import libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import sklearn
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.metrics import mean_squared_error
import pyarrow as pa
import pyarrow.parquet as pq
from bs4 import BeautifulSoup
from google.colab import drive
import re
import string
import nltk
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer, WordNetLemmatizer
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import MultinomialNB
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.metrics import classification_report
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer
from nltk.tokenize import WhitespaceTokenizer
import matplotlib.pyplot as plt
import seaborn as sns

from wordcloud import WordCloud
from collections import Counter

nltk.download('punkt')
nltk.download('stopwords')

import nltk
nltk.download('stopwords')

stop_words = set(nltk.corpus.stopwords.words('english'))

import unicodedata
from sklearn.pipeline import Pipeline
from scipy.sparse import hstack
import gensim
from gensim.models import Word2Vec
from sklearn.model_selection import StratifiedKFold
from tensorflow.keras.models import Model
import tensorflow as tf
from sklearn.model_selection import ParameterGrid
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.layers import Input, Dense, Dropout, LSTM, GRU, con
from tensorflow.keras.optimizers import Adam
from sklearn.feature_extraction.text import TfidfVectorizer
```

Data

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

In [ ]:
```
articles = pd.read_parquet('/content/drive/My Drive/Relevance/relevance_t
test = pd.read_parquet('/content/drive/My Drive/Relevance/relevance_test.
```

In [ ]:
```
# Training set
articles = pd.DataFrame(articles) # raw train dataset
articles_clean = pd.DataFrame(articles) # working training data

# Test set
test = pd.DataFrame(test) # raw train dataset
test_clean = pd.DataFrame(test) # working test data
```

In [ ]:
```
datasetShapeO, datasetShapeR = articles.shape
print("number of observations: ", datasetShapeO)
print("number of columns: ", datasetShapeR)
```

In [ ]:
```
# Checking column types and nulls
articles.info()
```

In [ ]:
```
print(articles_clean['body'].iloc[1])
```

Columns to clean

In [ ]:
```
columns_to_clean = ['body', 'title', 'description', 'narrative', 'topic_t
columns_to_clean_test = ['body', 'title', 'description', 'narrative', 'to
```

In [ ]:
```
print(articles_clean['body'].iloc[1])
```

Pipeline - cleaning data

In [ ]:
```
#Defining the classes for the pipeline

class Lowercase:
    def __init__(self, columns_to_clean):
        self.columns_to_clean = columns_to_clean

    def fit(self, X, y=None):
        return self

    def transform(self, X):
        X_copy = X.copy()
        for column in self.columns_to_clean:
            if column in X_copy.columns:
                X_copy[column] = X_copy[column].str.lower()
        return X_copy
```

```
    def fit_transform(self, X, y=None):
        return self.fit(X, y).transform(X)

class RemovingHTML:
    def __init__(self, column):
        self.column = column

    def fit(self, X, y=None):
        return self

    def transform(self, X):
        X_copy = X.copy()
        X_copy[self.column] = X_copy[self.column].apply(self.remove_html)
        return X_copy

    def fit_transform(self, X, y=None):
        return self.transform(X)

    def remove_html(self, text):
        if text is None:
            return ""
        soup = BeautifulSoup(text, 'html.parser')
        return soup.get_text()

class RemovingOtherStylingSymbols:
    def __init__(self, columns_to_clean):
        self.columns_to_clean = columns_to_clean

    def fit(self, X, y=None):
        return self

    def transform(self, X):
        X_copy = X.copy()
        for column in self.columns_to_clean:
            if column in X_copy.columns:
                X_copy[column] = X_copy[column].astype(str).apply(self.re
                X_copy[column] = X_copy[column].apply(self.remove_unicode
        return X_copy

    def fit_transform(self, X, y=None):
        return self.fit(X, y).transform(X)

    def remove_newlines_tabs(self, text):
        if pd.isnull(text):
            return ""
        text = text.replace('\\n', ' ').replace('\\t', ' ').replace('\n',
        return text

    def remove_unicode(self, text):
        if pd.isnull(text):
            return ""
        text = unicodedata.normalize("NFKD", text)
        return text

class RemovingPunctuation:
    def __init__(self, columns_to_clean):
```

```
        self.columns_to_clean = columns_to_clean

    def fit(self, X, y=None):
        return self

    def transform(self, X):
        X_copy = X.copy()
        for column in self.columns_to_clean:
            if column in X_copy.columns:
                X_copy[column] = X_copy[column].astype(str).str.replace('
        return X_copy

    def fit_transform(self, X, y=None):
        return self.fit(X, y).transform(X)

class RemoveStopwords:
    def __init__(self, columns_to_clean, stop_words):
        self.columns_to_clean = columns_to_clean
        self.stop_words = stop_words

    def fit(self, X, y=None):
        return self

    def transform(self, X):
        X_copy = X.copy()
        for column in self.columns_to_clean:
            if column in X_copy.columns:
                X_copy[column] = X_copy[column].apply(lambda text: ' '.jo
        return X_copy

    def fit_transform(self, X, y=None):
        return self.fit(X, y).transform(X)

class Lemmatization:
    def __init__(self, columns_to_clean):
        self.columns_to_clean = columns_to_clean
        self.wspt = WhitespaceTokenizer()
        self.wnl = WordNetLemmatizer()
        nltk.download('wordnet')
        nltk.download('omw-1.4')

    def lemmatize(self, text):
        return ' '.join([self.wnl.lemmatize(w) for w in self.wspt.tokeniz

    def fit(self, X, y=None):
        return self

    def transform(self, X):
        X_copy = X.copy()
        for column in self.columns_to_clean:
            if column in X_copy.columns:
                X_copy[column] = X_copy[column].astype(str).apply(self.le
        return X_copy

    def fit_transform(self, X, y=None):
        return self.fit(X, y).transform(X)
```

In [ ]:
```
pipeline = Pipeline([
    ('Lowercase', Lowercase(columns_to_clean)),
    ('RemovingHTML', RemovingHTML(column='body')),
    ('RemovingOtherStylingSymbols', RemovingOtherStylingSymbols(columns_t
    ('RemovingPunctuation', RemovingPunctuation(columns_to_clean)),
    ('RemoveStopwords', RemoveStopwords(columns_to_clean, stop_words)),
    ('Lemmatization', Lemmatization(columns_to_clean))
])

# training data
cleaned_articles = articles_clean.copy()
cleaned_articles = pipeline.fit_transform(cleaned_articles)

# test data
cleaned_articles_test = test.copy()
cleaned_articles_test = pipeline.transform(cleaned_articles_test)
```

In [ ]:
```
cleaned_articles.info()
cleaned_articles_test.info()
```

EDA

EDA re: 'Judgement'

In [ ]:
```
# Printing the count of each class
judgement_counts = cleaned_articles['judgement'].value_counts()
print("Count of each class in the 'judgement' column:")
print(judgement_counts)

# Visualizing the distribution of'judgement' column
plt.figure(figsize=(8, 6))
sns.countplot(x='judgement', data=cleaned_articles)
plt.title('Distribution of Judgement - cleaned_articles')
plt.xlabel('Judgement')
plt.ylabel('Count')
plt.show()
```

EDA re: 'Body'

In [ ]:
```
# Calculating the length of each bit of text in the 'body' column
cleaned_articles['body_length'] = cleaned_articles['body'].apply(len)

# Distribution of text lengths
plt.figure(figsize=(10, 6))
sns.histplot(cleaned_articles['body_length'], kde=True)
plt.title('Distribution of Text Lengths in the "body" Column')
plt.xlabel('Text Length')
plt.ylabel('Count')
plt.show()
```

In [ ]:
```
# Basic statistics of text lengths
body_length_stats = cleaned_articles['body_length'].describe()

print("Basic statistics of text lengths in the 'body' column:")
print(body_length_stats)
```

In [ ]:
```
# Number of words in the 'body' column
wt_words = cleaned_articles['body'].str.split().apply(len)

print(" words in the 'body' column:")
print(wt_words.sum())
```

In [ ]:
```
# Concatenating all text data from the 'body' column and tokenizing the c
all_text = ' '.join(cleaned_articles['body'].dropna())

words = nltk.word_tokenize(all_text)

# Frequency of each word
word_freq = Counter(words)

# Top 50 most common words
print("Frequency of the top 50 most common words in the 'body' column:")
print(word_freq.most_common(50))
```

EDA re: 'topic_id'

In [ ]:
```
# Number of unique categories of 'topic_id'
num_categories = cleaned_articles['topic_id'].nunique()

# Print the number of unique categories
print("Number of categories of 'topic_id':", num_categories)
```

Word Frequency in body in each topic

wordcloud

In [ ]:
```
# Grouping the data by 'topic_id' and concatenate text data within each g
grouped_data = cleaned_articles.groupby('topic_id')['body'].apply(' '.joi

# Word clouds for each group
for topic_id, text in grouped_data.items():
    wordcloud = WordCloud(width=800, height=400, background_color='white'
    plt.figure(figsize=(10, 6))
    plt.imshow(wordcloud, interpolation='bilinear')
    plt.title(f'Word Cloud for Topic ID {topic_id}')
    plt.axis('off')
    plt.show()
```

Word Frequency Counts

In [ ]:
```
word_counts = {}

for topic_id, text in grouped_data.items():
    words = text.split()
    word_freq = Counter(words)

    # Word frequencies in descending order
    sorted_word_freq = sorted(word_freq.items(), key=lambda x: x[1], reve

    #  Sorted word frequencies for the topic_id
    word_counts[topic_id] = sorted_word_freq

# Most frequent words for each 'topic_id' category
for topic_id, sorted_word_freq in word_counts.items():
    print(f"Most frequent words for Topic ID {topic_id}:")
    for word, freq in sorted_word_freq[:10]:
        print(f"{word}: {freq}")
    print()
```

Combine Columns

In [ ]:
```
def combine_columns(row):
    return ' '.join(str(item) if item is not None else '' for item in row

cleaned_articles['article_combined'] = cleaned_articles[['title', 'body',
cleaned_articles['query_combined'] = cleaned_articles[['description', 'na

cleaned_articles_test['article_combined'] = cleaned_articles_test[['title
cleaned_articles_test['query_combined'] = cleaned_articles_test[['descrip
```

In [ ]:
```
print(cleaned_articles['article_combined'].iloc[1])
```

Preprocessing

Term Frequency - Inverse Document Frequency (TF-IDF)

In [ ]:
```
tfidf_vectorizer_article = TfidfVectorizer(stop_words='english', max_feat
tfidf_vectorizer_query = TfidfVectorizer(stop_words='english', max_featur

article_combined_tfidf = tfidf_vectorizer_article.fit_transform(cleaned_a
query_combined_tfidf = tfidf_vectorizer_query.fit_transform(cleaned_artic

article_combined_tfidf_test = tfidf_vectorizer_article.transform(cleaned_
query_combined_tfidf_test = tfidf_vectorizer_query.transform(cleaned_arti

X_train_tfidf = hstack([article_combined_tfidf, query_combined_tfidf]) #s
X_test_tfidf = hstack([article_combined_tfidf_test, query_combined_tfidf_
```

Word Embedding

In [ ]:
```
def tokenize_texts(text_series):
    return [text.split() for text in text_series]

def get_average_embedding(tokenized_text, model_word2vec):
    embedding_size = model_word2vec.vector_size
    embeddings = [model_word2vec.wv[word] for word in tokenized_text if w
    if embeddings:
        return np.mean(embeddings, axis=0)
    else:
        return np.zeros(embedding_size)

tokenized_articles = tokenize_texts(cleaned_articles['article_combined'].
tokenized_queries = tokenize_texts(cleaned_articles['query_combined'].ast
tokenized_articles_test = tokenize_texts(cleaned_articles_test['article_c
tokenized_queries_test = tokenize_texts(cleaned_articles_test['query_comb

# Combined tokenized articles and queries for Word2Vec training
tokenized_texts = tokenized_articles + tokenized_queries
tokenized_texts_test = tokenized_articles_test + tokenized_queries_test
```

In [ ]:
```
# Training Word2Vec model
# note: we are using vector_size=50 to speed up compution

model_word2vec = Word2Vec(sentences=tokenized_texts, vector_size=50, wind
model_word2vec.train(tokenized_texts, total_examples=len(tokenized_texts)
```

In [ ]:
```
# Embeddings for articles and queries in training and test set.
articles_embeddings = np.array([get_average_embedding(text, model_word2ve
queries_embeddings = np.array([get_average_embedding(text, model_word2vec

articles_embeddings_test = np.array([get_average_embedding(text, model_wo
queries_embeddings_test = np.array([get_average_embedding(text, model_wor
```

In [ ]:
```
print("Shape of articles_embeddings:", articles_embeddings.shape)
print("Shape of queries_embeddings:", queries_embeddings.shape)
print("Shape of articles_embeddings_test:", articles_embeddings_test.shap
print("Shape of queries_embeddings_test:", queries_embeddings_test.shape)
```

In [ ]:
```
# Combining the vectorized features from both text columns into a single

X_train_Word = np.concatenate((articles_embeddings, queries_embeddings),

X_test_Word = np.concatenate((articles_embeddings_test, queries_embedding
```

Models

Standard Baseline

For the standard machine learning baseline, logistic regression, multinomial naive bayes, support vector machine and random forest classifier models were developed. Both TfIDf and Word2Vec were used for separate models.

In [ ]:
```
class MultiModelTrainer:
    def __init__(self, models, n_splits=5):
        self.models = models
        self.n_splits = n_splits
        self.best_model = None
        self.model_accuracies = {}

    def train_and_evaluate(self, X, y):
        skf = StratifiedKFold(n_splits=self.n_splits, shuffle=True, rando

        for name, model in self.models:
            fold_accuracies = []

            # Dividing training and validation set with Stratified k-fold
            for train_index, val_index in skf.split(X, y):
                X_train, X_val = X[train_index], X[val_index]
                y_train, y_val = y[train_index], y[val_index]

                # fit & evaluate
                model.fit(X_train, y_train)
                score = model.score(X_val, y_val)
                fold_accuracies.append(score)

            # Mean accuracy across folds for the current model
            mean_accuracy = np.mean(fold_accuracies)
            self.model_accuracies[name] = mean_accuracy
            print(f"{name}: Mean accuracy = {mean_accuracy}")

        # Best model based on mean accuracy
        best_model_name = max(self.model_accuracies, key=self.model_accur
        self.best_model = [m for n, m in self.models if n == best_model_n
        print(f"\nBest model: {best_model_name} with accuracy = {self.mod

    def fit_best_model_and_predict(self, X_train, y_train, X_test):
        if not self.best_model:
            raise ValueError("No model has been trained yet.")

        # Fit the best model on the entire training dataset
        self.best_model.fit(X_train, y_train)

        # Predicting on the test dataset
        predictions = self.best_model.predict(X_test)

        return predictions
```

Tfdif

In [ ]:
```
y = articles_clean['judgement'].values

models = [
    ('Logistic Regression', LogisticRegression(random_state=42)),
    ('Multinomial Naive Bayes', MultinomialNB()),
    ('Random Forest', RandomForestClassifier(random_state=42)),
    ('Support Vector Machine', SVC(kernel='rbf', gamma='scale', random_st
]

trainer = MultiModelTrainer(models=models, n_splits=5)
trainer.train_and_evaluate(X_train_tfidf, y)
predictions_tfidf = trainer.fit_best_model_and_predict(X_train_tfidf, y,
```

Word2Vec embeddings

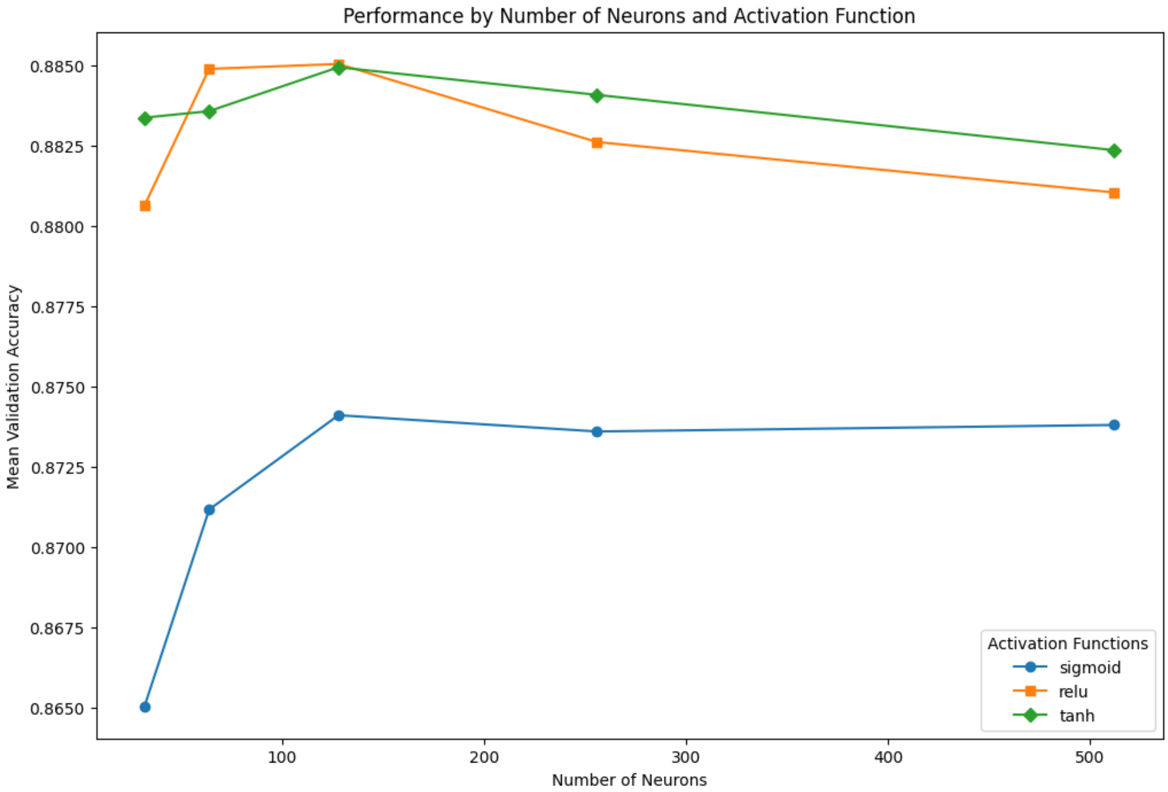
In [ ]:
```
y = articles_clean['judgement'].values

models = [
    ('Logistic Regression', LogisticRegression(random_state=42)),
    ('Random Forest', RandomForestClassifier(random_state=42)),
    ('Support Vector Machine', SVC(kernel='rbf', gamma='scale', random_st
]

trainer = MultiModelTrainer(models=models, n_splits=5)
trainer.train_and_evaluate(X_train_Word, y)
predictions_tfidf = trainer.fit_best_model_and_predict(X_train_Word, y, X
```

In [ ]:
```
#best model prediction to csv
submission = pd.DataFrame({
    'id': test_clean['doc_id'],
    'judgement': predictions_tfidf
})
submission.to_csv('standardbl_tfidf.csv', index=False)
```

3Layer NN Baseline

The activation type in the output layer uses a sigmoid function. Then, accuracy is calculated using binary cross-entropy and weights and biases are updated with the Adaptive Moment Estimation optimiser (Géron, 2019, p135 & p356).

Our optimal model used relu and 256 neurons over 10 epochs. Sigmoid and relu plateau at 128 neurons, whilst tanh peaks at 128 neurons. From 256 neurons, relu and sigmoid decrease in performance. Despite tanh performing better than relu at 32 neurons, relu performs better overall with an optimal number of neurons. The sigmoid activation function returns the poorest accuracy score.



In [ ]:
```
class BasicNeuralNetwork(Model):
    def __init__(self, input_shape_article, input_shape_query, neurons, a
        super(BasicNeuralNetwork, self).__init__()
        self.input_shape_article = input_shape_article
        self.input_shape_query = input_shape_query
        self.neurons = neurons
        self.activation = activation
        self.build_model()

    def build_model(self):
        input_article = Input(shape=(self.input_shape_article,), name='Ar
        input_query = Input(shape=(self.input_shape_query,), name='Query_
        concatenated_inputs = concatenate([input_article, input_query])
        hidden_layer = Dense(self.neurons, activation=self.activation)(co
        output = Dense(1, activation='sigmoid')(hidden_layer)
        self.model = Model(inputs=[input_article, input_query], outputs=o

    def compile(self, optimizer='adam', loss='binary_crossentropy', metri
        self.model.compile(optimizer=optimizer, loss=loss, metrics=metric

    def fit(self, x, y, **kwargs):
        return self.model.fit(x, y, **kwargs)

    def evaluate(self, x, y, **kwargs):
        return self.model.evaluate(x, y, **kwargs)

    def predict(self, x, **kwargs):
        return self.model.predict(x, **kwargs)
```

In [ ]:
```
# Set random seed for reproducibility
np.random.seed(42)
tf.random.set_seed(42)

# Parameters
n_splits = 5
epochs = 10
batch_size = 32
neurons_list = [32, 64, 128, 256, 512]
activation_functions = ['relu', 'tanh', 'sigmoid']
y_train = articles_clean['judgement'].values
skf = StratifiedKFold(n_splits=n_splits, shuffle=True, random_state=42)

# Performance for each activation function
performance_records = []

for neurons in neurons_list:
    for activation in activation_functions:
        fold_accuracies = []
        for train_index, val_index in skf.split(articles_embeddings, y_tr
            model = BasicNeuralNetwork(articles_embeddings.shape[1], quer
            model.compile(optimizer='adam', loss='binary_crossentropy', m
            early_stopping = EarlyStopping(monitor='val_loss', patience=3

            # Split the data
            articles_train_fold, articles_val_fold = articles_embeddings[
            queries_train_fold, queries_val_fold = queries_embeddings[tra
```

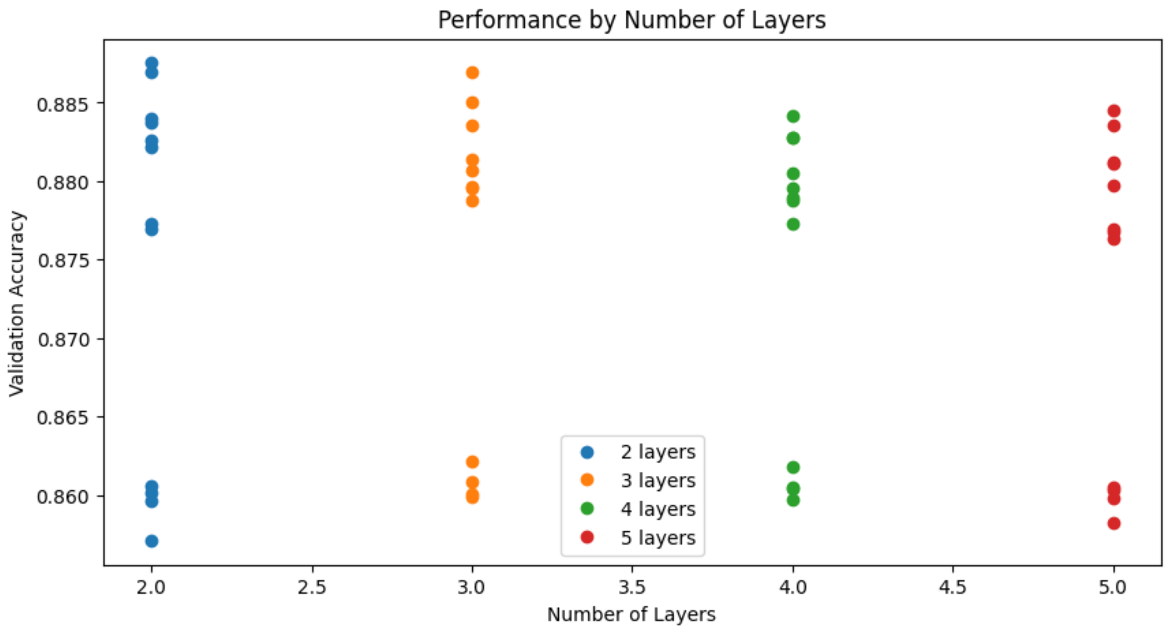
```
            y_train_fold, y_val_fold = y_train[train_index], y_train[val_

            model.fit([articles_train_fold, queries_train_fold], y_train_
                      validation_data=([articles_val_fold, queries_val_fo
                      epochs=epochs, batch_size=batch_size, callbacks=[ea
            val_accuracy = model.evaluate([articles_val_fold, queries_val
            fold_accuracies.append(val_accuracy)

        mean_val_accuracy = np.mean(fold_accuracies)
        performance_records.append((neurons, activation, mean_val_accurac

# Plotting performance
plt.figure(figsize=(12, 8))
markers = ['o', 's', 'D', '^', 'v', '>', '<', 'p', '*', 'h', 'x']
activations = list(set(record[1] for record in performance_records))
marker_dict = dict(zip(activations, markers[:len(activations)]))

for activation in activations:
    records = [record for record in performance_records if record[1] == a
    neurons = [record[0] for record in records]
    accuracies = [record[2] for record in records]

    plt.plot(neurons, accuracies, marker=marker_dict[activation], linesty

plt.xlabel('Number of Neurons')
plt.ylabel('Mean Validation Accuracy')
plt.title('Performance by Number of Neurons and Activation Function')
plt.legend(title="Activation Functions")
plt.show()

# Best configuration
best_record = max(performance_records, key=lambda x: x[2])
print(f"Best Configuration: Neurons: {best_record[0]}, Activation: {best_



# Final model
optimal_neurons = max(performance_records, key=lambda x: x[2])[0]
optimal_activation = max(performance_records, key=lambda x: x[2])[1]
print(f"Using optimal number of neurons: {optimal_neurons} and activation

final_model = BasicNeuralNetwork(articles_embeddings.shape[1], queries_em
final_model.compile(optimizer='adam', loss='binary_crossentropy', metrics
final_model.fit([articles_embeddings, queries_embeddings], y_train, epoch

# Predictions with the best model
predictions = final_model.predict([articles_test, queries_test])
binary_predictions_3L = (predictions >= 0.5).astype(int)

binary_predictions_3L
```

In [ ]:
```
binary_predictions_3L = np.squeeze(binary_predictions_3L)

submission = pd.DataFrame({
    'id': test_clean['doc_id'],
    'judgement': binary_predictions_3L.astype(int)
})

submission.to_csv('submission3L_word.csv', index=False)
```

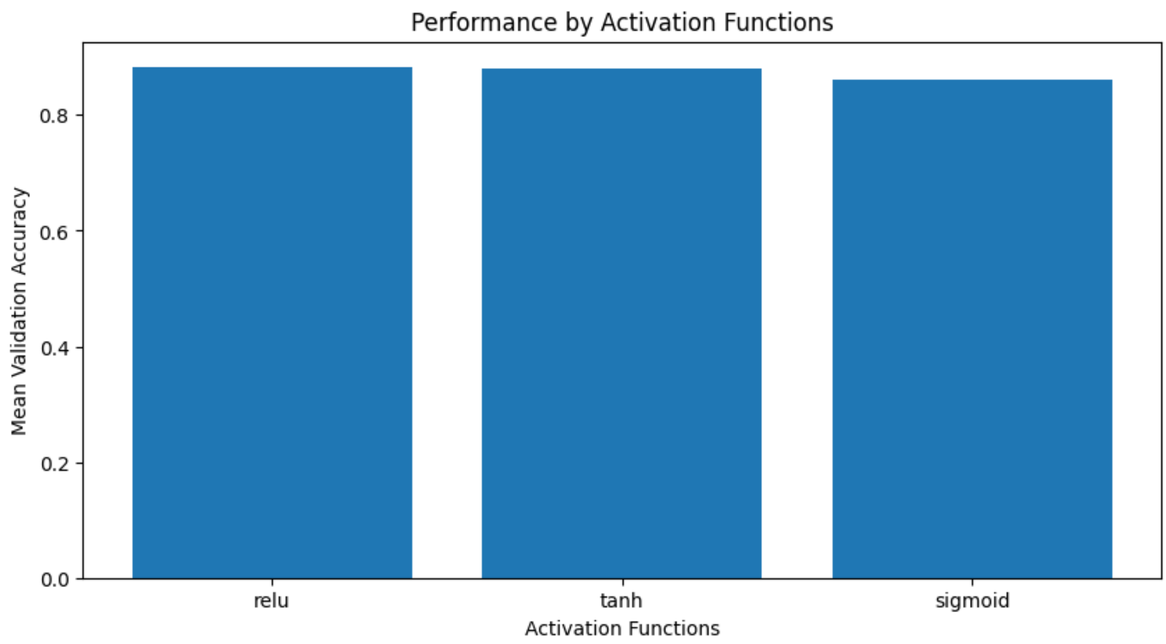
Deep NN models

For the deep neural network model, the articles and queries were fed into the model as two separate inputs. For this model, the number of layers explored were 3, 4, 5, 6 and 7 with 256 neurons using activations of relu, tanh and sigmoid.

The most optimum performance of the model occurs when there are 6 layers. There is a slight increase in performance from 5 to 6 layers and the performance decreases slightly when moving to 7 layers.



The most optimum performance of the model occurs when the relu activation function is used. The mean validation accuracy for the model is similar for relu, tanh and sigmoid activations.



The best configuration for the deep neural network model is 6 layers with 256 neurons, relu activation, initialiser of he_normal and a learning rate of 0.001.

In [ ]:
```
# Set random seed for reproducibility
np.random.seed(42)
tf.random.set_seed(42)

class DeepNNTrainer:
    def __init__(self, input_shape_article, input_shape_query, epochs=10,
        self.input_shape_article = input_shape_article
        self.input_shape_query = input_shape_query
        self.epochs = epochs
        self.batch_size = batch_size
        self.n_splits = n_splits
        self.performance_records = {}
        self.best_model = None
        self.best_accuracy = 0

    def build_model(self, num_layers, neurons_per_layer, activation, kern
        input_article = Input(shape=(self.input_shape_article,), name='Ar
        input_query = Input(shape=(self.input_shape_query,), name='Query_

        # First branch for articles
        x1 = Dense(neurons_per_layer, activation=activation, kernel_initi
        # Second branch for queries
        x2 = Dense(neurons_per_layer, activation=activation, kernel_initi

        # Combining the outputs from both branches
        combined = concatenate([x1, x2])

        # Adding additional layers based on num_layers
        for _ in range(num_layers - 1):
            combined = Dense(neurons_per_layer, activation=activation, ke

        output = Dense(1, activation='sigmoid')(combined)
```

```
        model = Model(inputs=[input_article, input_query], outputs=output
        model.compile(optimizer='adam', loss='binary_crossentropy', metri

        return model

    def regularizer(self, kernel_regularizer):
        if kernel_regularizer is None:
            return None
        elif kernel_regularizer == "l1":
            return regularizers.l1()
        elif kernel_regularizer == "l2":
            return regularizers.l2()
        elif kernel_regularizer == "l1l2":
            return regularizers.l1_l2()
        else:
            raise ValueError("Not a valid regularizer. Use 'l1', 'l2' or

    # Updated train_and_evaluate method to accept and handle two sets of
    def train_and_evaluate(self, articles_train, queries_train, y_train,
        X = np.zeros((articles_train.shape[0], 1))
        skf = StratifiedKFold(n_splits=self.n_splits, shuffle=True, rando

        for num_layers in num_layers_options:
            for neurons_per_layer in neurons_options:
                for activation in activations_options:
                    for kernel_initializer in kernel_initializer_options:
                        for kernel_regularizer in regularization_options
                            for learning_rate in learning_rate_options or
                                fold_accuracies = []
                                for train_index, val_index in skf.split(X
                                    # Split the data
                                    articles_train_fold, articles_val_fol
                                    queries_train_fold, queries_val_fold
                                    y_train_fold, y_val_fold = y_train[tr

                                    # Build and compile the model
                                    model = self.build_model(num_layers,
                                    optimizer = Adam(learning_rate=learni
                                    early_stopping = EarlyStopping(monito

                                    # Train the model
                                    model.compile(optimizer=optimizer, lo
                                    model.fit([articles_train_fold, queri
                                              validation_data=([articles_
                                              epochs=self.epochs, batch_s

                                    # Evaluate the model
                                    val_accuracy = model.evaluate([articl
                                    fold_accuracies.append(val_accuracy)

                                mean_val_accuracy = np.mean(fold_accuraci
                                self.performance_records[(num_layers, neu
                                if mean_val_accuracy > self.best_accuracy
                                    self.best_accuracy = mean_val_accurac
                                    self.best_model = model
                                print(f"Layers: {num_layers}, Neurons: {n
```

```
        self.best_config = max(self.performance_records, key=self.perform
        print(f"Best configuration: {self.best_config} with accuracy {sel

        # Retrain best model on all data
        best_num_layers, best_neurons, best_activation, best_initializer,
        self.best_model = self.build_model(best_num_layers, best_neurons,
        optimizer = Adam(learning_rate=best_learning_rate) if best_learni
        self.best_model.compile(optimizer=optimizer, loss='binary_crossen
        self.best_model.fit([articles_train, queries_train], y_train, epo

    def predict(self, articles_test, queries_test):
        if self.best_model:
            return self.best_model.predict([articles_test, queries_test])
        else:
            raise AttributeError("No best model to predict. Please run tr
```

In [ ]:

```
# Parameters
num_layers_options = [3, 4, 5, 6, 7]
neurons_options = [256]
activations_options = ['relu']
regularization_options = [None]
kernel_initializer_options = ['he_normal']
learning_rate_options = [0.001, 0.01, 0.1]

# initialize DeepNNTrainer
trainer = DeepNNTrainer(input_shape_article=articles_train_deep.shape[1],
                        input_shape_query=queries_train_deep.shape[1],
                        epochs=10, batch_size=32, n_splits=5)

trainer.train_and_evaluate(articles_train_deep, queries_train_deep, y_tra
                           num_layers_options, neurons_options, activatio
                           kernel_initializer_options, regularization_opt

# Binary predictions
predictions = trainer.predict(articles_test_deep, queries_test_deep)

# Binary format
binary_predictions_deep_nn = (predictions >= 0.5).astype(int).flatten()
```

In [ ]:

```
# Binary predictions to a CSV file
submission = pd.DataFrame({
    'id': test_clean['doc_id'],
    'judgement': binary_predictions_deep_nn
})
submission.to_csv('deepnn.csv', index=False)
```

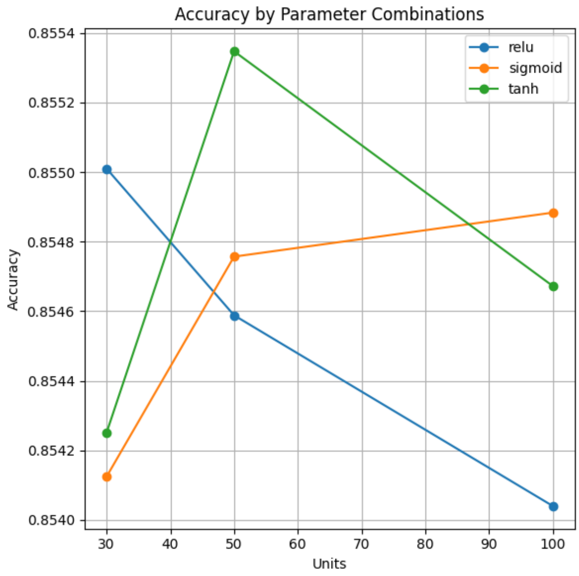
Exploring relu', 'leaky_relu', 'elu', 'swish' activations

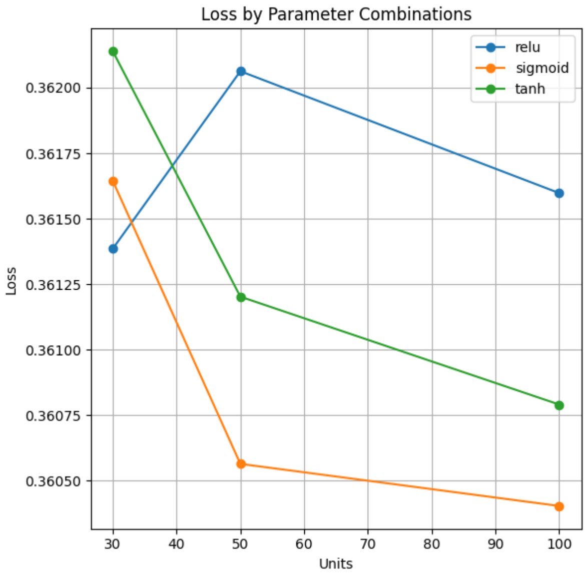
In [ ]:
```
# parameters
num_layers_options = [3, 4, 5, 6, 7]
neurons_options = [256]
activations_options = ['relu', 'leaky_relu', 'elu', 'swish']
regularization_options = [None]
kernel_initializer_options = ['he_normal']
learning_rate_options = [0.001]

# Instantiate the DeepNNTrainer
trainer = DeepNNTrainer(input_shape_article=articles_train_deep.shape[1],
                        input_shape_query=queries_train_deep.shape[1],
                        epochs=10, batch_size=32, n_splits=5)

# Train and evaluate the model
trainer.train_and_evaluate(articles_train_deep, queries_train_deep, y_tra
                           num_layers_options, neurons_options, activatio
                           kernel_initializer_options, regularization_opt

# Binary predictions
predictions = trainer.predict(articles_test_deep, queries_test_deep)

# Binary format
binary_predictions_deep_nn = (predictions >= 0.5).astype(int).flatten()
```

Complex NN Models

Wide and Deep

For the wide and deep model, the articles were added into the wide layer and queries were added into the deep layer. Two hidden layers were also added. The parameters explored were units of 30, 50 and 100; hidden layers of (30,), (50,), (100,), (30, 30), (50, 50) and (100, 100); activations relu, tanh and sigmoid and ouput activation of sigmoid. As illustrated by the charts below, the relu activation produces a higher accuracy and lower loss as compared to sigmoid and tanh, especially when the number of units increases.





In [ ]:
```
# Splitting the data into training and validation sets
X_train_wide, X_valid_wide, X_train_deep, X_valid_deep, y_train, y_valid
    articles_embeddings, queries_embeddings, y_train, test_size=0.2, rand
)

# Adapt normalization layers to training data
norm_wide_layer.adapt(X_train_wide)
norm_deep_layer.adapt(X_train_deep)

# Check the sizes of training and validation data
print("Sizes of Training Data:", len(X_train_wide), len(X_train_deep), le
print("Sizes of Validation Data:", len(X_valid_wide), len(X_valid_deep),
```

In [ ]:
```
# input layers for wide and deep inputs
input_articles = keras.layers.Input(shape=[50], name="wide_input")
input_queries = keras.layers.Input(shape=[50], name="deep_input")

# normalization layers
norm_wide_layer = keras.layers.Normalization(axis=None)
norm_deep_layer = keras.layers.Normalization(axis=None)
norm_wide = norm_wide_layer(input_articles)
norm_deep = norm_deep_layer(input_queries)

# deep layers
hidden1 = keras.layers.Dense(30, activation="relu")(norm_deep)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
```

```
# Concatenation wide and deep outputs
concat = keras.layers.Concatenate()([norm_wide, hidden2])
output = keras.layers.Dense(1, activation='sigmoid')(concat)

class WideAndDeepModel(keras.Model):
    def __init__(self, units=30, activation="relu", output_activation="si
        super().__init__(**kwargs)
        self.norm_layer_wide = keras.layers.Normalization()
        self.norm_layer_deep = keras.layers.Normalization()
        self.hidden_layers = [keras.layers.Dense(units, activation=activa
        self.main_output = keras.layers.Dense(1, activation=output_activa

    def call(self, inputs):
        input_wide, input_deep = inputs
        norm_wide = self.norm_layer_wide(input_wide)
        norm_deep = self.norm_layer_deep(input_deep)
        deep = norm_deep
        for layer in self.hidden_layers:
            deep = layer(deep)
        concat = keras.layers.concatenate([norm_wide, deep])
        main_output = self.main_output(concat)
        return main_output

param_grid = {
    'units': [30, 50, 100],
    'activation': ['relu', 'tanh', 'sigmoid'],
    'output_activation': ['sigmoid'],
    'hidden_layers': [(30,), (50,), (100,), (30, 30), (50, 50), (100, 100
}

results = []
#callbacks
early_stopping = EarlyStopping(patience=3, restore_best_weights=True)

# parameter combinations
for params in ParameterGrid(param_grid):
    print("Trying configuration:", params)

    #initialize and compile
    model = WideAndDeepModel(**params)
    model.compile(loss="binary_crossentropy", optimizer="adam", metrics=[

    #fit
    history = model.fit(
        [X_train_wide, X_train_deep], y_train, epochs=20,
        validation_data=([X_valid_wide, X_valid_deep], y_valid),
        callbacks=[early_stopping], verbose=0
    )

    #evaluate
    loss, accuracy = model.evaluate([X_valid_wide, X_valid_deep], y_valid
    results.append({'params': params, 'accuracy': accuracy, 'history': hi


best_model = max(results, key=lambda x: x['accuracy'])
print("Best model configuration:", best_model['params'])
print("Best model accuracy:", best_model['accuracy'])
```

```
#Plot
pd.DataFrame(history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.show()
```

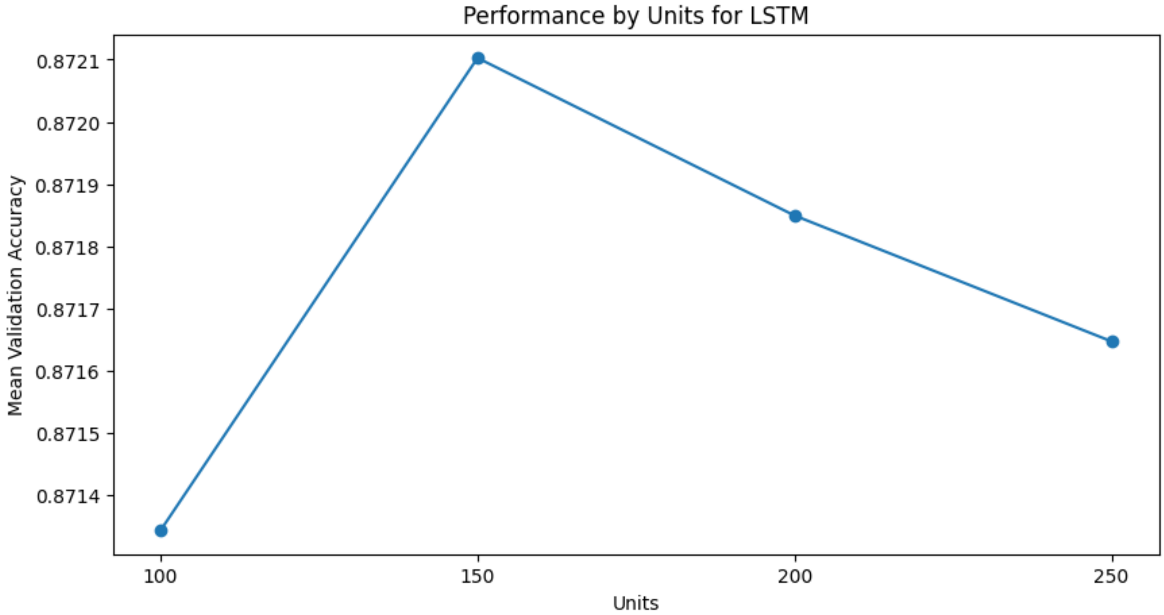
In [ ]:

```
# parameter configurations and corresponding accuracy and loss results to
params_list = [result['params'] for result in results]
accuracies = [result['accuracy'] for result in results]
losses = [result['history'].history['loss'][-1] for result in results]

results_df = pd.DataFrame({
    'Units': [params['units'] for params in params_list],
    'Activation': [params['activation'] for params in params_list],
    'Accuracy': accuracies,
    'Loss': losses
})

# results by activation function and units. mean acc & loss
grouped_results = results_df.groupby(['Activation', 'Units']).mean().rese

# plots
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
for activation, group in grouped_results.groupby('Activation'):
    plt.plot(group['Units'], group['Accuracy'], label=activation, marker=
plt.title('Accuracy by Parameter Combinations')
plt.xlabel('Units')
plt.ylabel('Accuracy')
plt.grid(True)
plt.legend()


plt.subplot(1, 2, 2)
for activation, group in grouped_results.groupby('Activation'):
    plt.plot(group['Units'], group['Loss'], label=activation, marker='o')
plt.title('Loss by Parameter Combinations')
plt.xlabel('Units')
plt.ylabel('Loss')
plt.grid(True)
plt.legend()

plt.tight_layout()
plt.show()
```

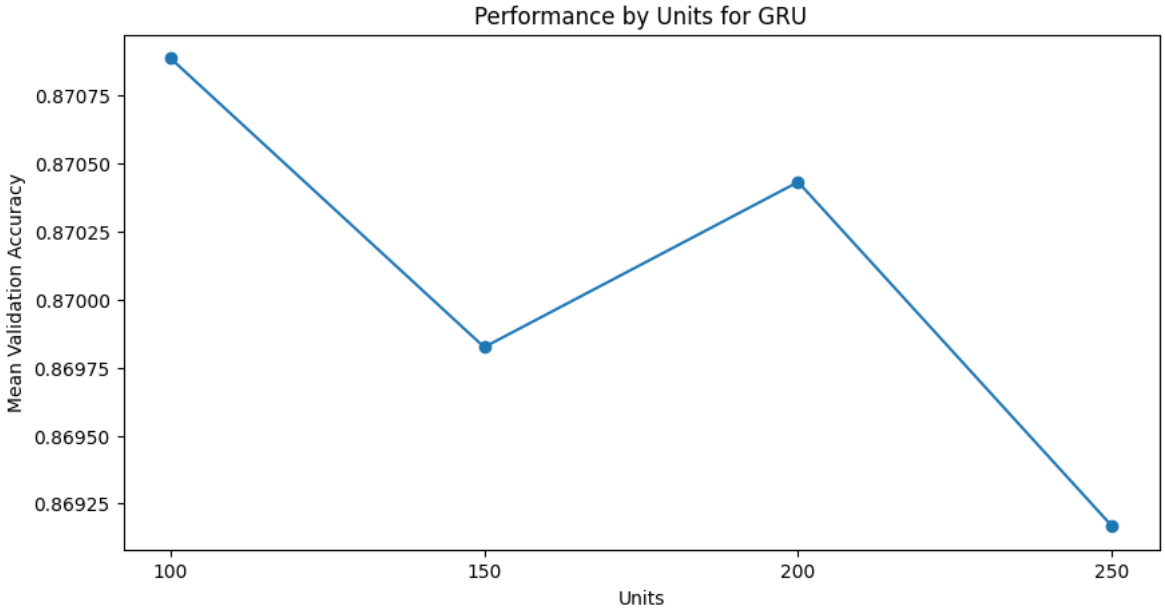
RNN

The RNN models developed include LSTM and GRU. The units explored in the model were 50, 100, 150, 200 and 250 with activation options of tahn and sigmoid. The best configuration was the LSTM model with 150 units using the tanh activation.

The highest accuracy was with 150 units, after which accuracy decreases. This indicates that adding more units did not improve the model's performance.



The highest accuracy was with 100 units, after which accuracy fluctuated and remained low. This indicates that adding more units did not improve the model's performance.



In [ ]:
```
np.random.seed(42)
tf.random.set_seed(42)
```

```
class RNNTrainer:
    def __init__(self, input_shape_article, input_shape_query, output_uni
        self.input_shape_article = input_shape_article
        self.input_shape_query = input_shape_query
        self.output_units = output_units
        self.epochs = epochs
        self.batch_size = batch_size
        self.n_splits = n_splits
        self.performance_records = {}
        self.best_model = None
        self.best_accuracy = 0

    def build_model(self, rnn_unit_type='LSTM', units=50, activation='tan
        input_article = Input(shape=self.input_shape_article, name='Artic
        input_query = Input(shape=self.input_shape_query, name='Query_Inp

        if rnn_unit_type == 'LSTM':
            article_rnn = LSTM(units, activation=activation, recurrent_ac
            query_rnn = LSTM(units, activation=activation, recurrent_acti
        elif rnn_unit_type == 'GRU':
            article_rnn = GRU(units, activation=activation, recurrent_act
            query_rnn = GRU(units, activation=activation, recurrent_activ

        concatenated = concatenate([article_rnn, query_rnn])
        output = Dense(self.output_units, activation='sigmoid')(concatena

        model = Model(inputs=[input_article, input_query], outputs=output
        model.compile(optimizer='adam', loss='binary_crossentropy', metri
        return model

    def train_and_evaluate(self, articles_train, queries_train, y_train,
        X_dummy = np.zeros((articles_train.shape[0], 1))
        skf = StratifiedKFold(n_splits=self.n_splits, shuffle=True, rando

        for rnn_unit in rnn_unit_options:
            for units in units_options:
                for activation, recurrent_activation in activations_optio
                    fold_accuracies = []
                    for train_index, val_index in skf.split(X_dummy, y_tr
                        X_article_train_fold, X_article_val_fold = articl
                        X_query_train_fold, X_query_val_fold = queries_tr
                        y_train_fold, y_val_fold = y_train[train_index],

                        model = self.build_model(rnn_unit_type=rnn_unit,
                                                 activation=activation, r
                        early_stopping = EarlyStopping(monitor='val_loss'
                        model.fit([X_article_train_fold, X_query_train_fo
                                  validation_data=([X_article_val_fold, X
                                  epochs=self.epochs, batch_size=self.bat

                        val_accuracy = model.evaluate([X_article_val_fold
                        fold_accuracies.append(val_accuracy)

                    mean_val_accuracy = np.mean(fold_accuracies)
                    self.performance_records[(rnn_unit, units, activation
                    if mean_val_accuracy > self.best_accuracy:
                        self.best_accuracy = mean_val_accuracy
```

```
                        self.best_model = model
                        self.best_config = (rnn_unit, units, activation)

        print(f"Best configuration: {self.best_config} with accuracy {sel

    def predict(self, articles_test, queries_test):
        if self.best_model:
            return self.best_model.predict([articles_test, queries_test])
        else:
            raise ValueError("No best model to predict. Please run train_

    def save_model(self, file_path):
        if self.best_model:
            self.best_model.save(file_path)
        else:
            raise ValueError("No best model to save. Please run train_and

    def load_model(self, file_path):
        self.best_model = tf.keras.models.load_model(file_path)

    def plot_performance_by_units(self, rnn_unit):
    # Plotting performance by the number of units
        unit_performance = {key[1]: val for key, val in self.performance_
        units = list(unit_performance.keys())
        performances = list(unit_performance.values())

        plt.figure(figsize=(10, 5))
        plt.plot(units, performances, marker='o')
        plt.title(f'Performance by Units for {rnn_unit}')
        plt.xlabel('Units')
        plt.ylabel('Mean Validation Accuracy')
        plt.xticks(units)
        plt.show()

    def plot_performance_by_rnn_unit(self, units):
    # Plotting performance by rnn unit type
        rnn_unit_performance = {key[0]: val for key, val in self.performa
        rnn_units = list(rnn_unit_performance.keys())
        performances = list(rnn_unit_performance.values())

        plt.figure(figsize=(10, 5))
        plt.bar(rnn_units, performances, color='skyblue')
        plt.title(f'Performance by RNN Unit with {units} Units')
        plt.xlabel('RNN Unit Type')
        plt.ylabel('Mean Validation Accuracy')
        plt.xticks(rnn_units)
        plt.show()
```

In [ ]:
```
# Correcting the shape
articles_train_rnn = np.expand_dims(articles_embeddings, axis=1)
queries_train_rnn = np.expand_dims(queries_embeddings, axis=1)

articles_test_rnn = np.expand_dims(articles_embeddings_test, axis=1)
queries_test_rnn = np.expand_dims(queries_embeddings_test, axis=1)

print(articles_train_rnn.shape)
print(queries_train_rnn.shape)
```

In [ ]:
```
# Loading data and labels
articles_train = np.load('articles_embeddings.npy')
queries_train = np.load('queries_embeddings.npy')
articles_test = np.load('articles_embeddings_test.npy')
queries_test = np.load('queries_embeddings_test.npy')
y_train = np.load('labels.npy')


articles_train_rnn = np.expand_dims(articles_train, axis=1)
queries_train_rnn = np.expand_dims(queries_train, axis=1)
articles_test_rnn = np.expand_dims(articles_test, axis=1)
queries_test_rnn = np.expand_dims(queries_test, axis=1)

#trainer with new input shape
trainer = RNNTrainer(
    input_shape_article=articles_train_rnn.shape[1:],
    input_shape_query=queries_train_rnn.shape[1:],
    output_units=1,
    epochs=10,
    batch_size=32,
    n_splits=5
)

# hyperparameters
rnn_unit_options = ['LSTM', 'GRU']
units_options = [100, 150, 200, 250]
activations_options = [('tanh', 'sigmoid')]


# Train and evaluate
trainer.train_and_evaluate(articles_train_rnn, queries_train_rnn, y_train
trainer.save_model('best_rnn_model.h5')

# can load the best model below, if needed
# trainer.load_model('best_rnn_model.h5')

#Predict with best
predictions = trainer.predict(articles_test_rnn, queries_test_rnn)
binary_predictions_rnn = (predictions >= 0.5).astype(int) # pred to binar

# plot
trainer.plot_performance_by_units(rnn_unit='LSTM')
trainer.plot_performance_by_units(rnn_unit='GRU')
```

In [ ]:
```
# results to csv

binary_predictions_1d = np.squeeze(binary_predictions_rnn)


submission = pd.DataFrame({
    'id': test_clean['doc_id'],
    'judgement': binary_predictions_1d
})

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

In [ ]:
```
articles_train = np.load('articles_embeddings.npy')
queries_train = np.load('queries_embeddings.npy')
articles_test = np.load('articles_embeddings_test.npy')
queries_test = np.load('queries_embeddings_test.npy')
y_train = np.load('labels.npy')

articles_train_rnn = np.expand_dims(articles_train, axis=1)
queries_train_rnn = np.expand_dims(queries_train, axis=1)
articles_test_rnn = np.expand_dims(articles_test, axis=1)
queries_test_rnn = np.expand_dims(queries_test, axis=1)

trainer = RNNTrainer(
    input_shape_article=articles_train_rnn.shape[1:],
    input_shape_query=queries_train_rnn.shape[1:],
    output_units=1,
    epochs=10,
    batch_size=32,
    n_splits=5
)


rnn_unit_options = ['LSTM', 'GRU']
units_options = [50, 100, 150, 200]
activations_options = [('tanh', 'sigmoid')]


trainer.train_and_evaluate(articles_train_rnn, queries_train_rnn, y_train
trainer.save_model('best_rnn_model.h5')

# can load the best model here, if needed
# trainer.load_model('best_rnn_model.h5')


predictions = trainer.predict(articles_test_rnn, queries_test_rnn)
binary_predictions_rnn = (predictions >= 0.5).astype(int)


trainer.plot_performance_by_units(rnn_unit='LSTM')
trainer.plot_performance_by_units(rnn_unit='GRU')
```

Results

Across various neural network architectures, our models received good accuracy scores. Given that the training data itself required a lot of preprocessing and even included incorrect relevancy scores for some instances, each neural network model adapted well to identify patterns, intricacies and relationships in the news articles dataset. Where possible, multiple iterations (beyond the epoch iterations within the models themselves) were run for each neural network. This ensured that we could capture the mean accuracy score of any given architecture and potential mitigate randomness, especially as each run would take different splits of the data.

Across the board, the relu activation function performed consistently well, namely for the three-layer architecture, the deep neural networks and the wide and deep neural networks. Compared to other activation functions, it is more simplistic with an aggressive decision line and no maximum output value; this helped to mitigate the vanishing gradient problem in which learning rates of parameters become increasingly small.

Consistently amongst different neural networks, we found that the optimal number of neurons per layer was between 128 and 256. Whilst initially it may seem that adding more neurons is an intuitive way to compute more features, we found that by having too many unimportant features in training, the models would begin to overfit and learning would plateau or decrease. For instance, in the three-layer neural network, 256 neurons resulted in 88.697% accuracy whilst 128 neurons had 88.50%.

In our standard baseline model classifiers, we tested both word2vec inputs and TfIDf vectorised inputs. We found that TfIDf inputs performed poorly and learned too slowly because of the large amount of inputs, this may be because of the extremely large dictionary (corpus) it had to work with, which was the length of every unique word in the cleaned database. With word2vec, however, our best accuracy score was captured at 89.02%. This was with ensemble learning methods using a random forest classifier, which is a collection of multiple decision trees trained on separate splits in the data then joined together.

We tested a deep neural network with six and seven layers. Our six-layer model performed slightly better than our seven layer, with 88.55% and 88.52% respectively. However, our wide and deep models with relu and 100 layers and sigmoid and 30 layers scored 85.63% and 85.60% respectively.

Our complex recurrent neural networks performed similarly but marginally worse than above, with LSTM scoring a mean accuracy of 87.41%.

Model name	Iteration	Configuration / Best Performance	Accuracy Score	Mean	Standard Deviation
Standard Baseline	1	Word2Vec - Random Forest	0.89017	-	-
	2	Word2Vec - Random Forest	0.89139	-	-
				0.8908	0.0009
	1	TfIDf - Random Forest	0.88511	-	-
	2	TfIDf - Support Vector Machine	0.88646	-	-
				0.8851	0.0010
3L NN BL	1	Neurons: 128, Activation: relu	0.88504	-	-
	2	Neurons: 256, Activation: relu	0.88697	-	-
				0.8860	0.0014
Deep NN	1	Layers: 6, Neurons: 256, Activation: 'relu', Initialiser: 'he_normal', Regularisation: None, Learning Rate: 0.001	0.88551	-	-
	2	Layers: 7, Neurons: 256, Activation: 'relu', Initialiser: 'he_normal', Regularisation: None, Learning Rate: 0.001	0.88526	-	-
				0.8854	0.0002
RNN	1	LSTM, Units: 150, Activation: 'tanh'	0.8721	-	-
	2	LSTM, Units: 150, Activation: 'tanh'	0.8741	-	-
				0.8741	0.0028
Wide and Deep	1	Activation': 'relu', Hidden layers: (100, 100), Output activation: 'sigmoid', Units: 30	0.85628	-	-
	2	Activation': 'sigmoid', Hidden layers: (30,), Output activation: 'sigmoid', Units: 50	0.85602	-	-
				0.8562	0.0002

Model name	Kaggle Score
Standard Baseline	0.87305
3L NN	0.86950
Deep NN	0.87114
RNN	0.86814
Wide and Deep	0.87223

Summary

The best performing model for this task is standard ML baseline random forest model. This model used Word2Vec. The model's mean performance was 89.1% and scored 87.3% on Kaggle. This Kaggle score is at the top of the leader board.

While the standard ML model performed better than neural networks, it does not necessarily imply that this type of model is ideally suited for text relevance tasks. Various factors could have influenced the results such as the dataset being too small and data linearity. Therefore, the recommended model is the 3-layer baseline model which had a mean accuracy of 88.6% and a Kaggle score of 87%. The score is within the top 5% of the leaderboard on Kaggle.

Making financial investments in the Pro version of Google Colab, will allow for the development of more complex and computationally heavy models such as BERT. Using such pretrained models can assist in increasing the performance of the model (Htait, 2021).

References

Azzopardi, L. (2024). 'Processing Text Features', CS985: Machine Learning for Data Analytics. Available at: https://classes.myplace.strath.ac.uk/course/view.php?id=22388 (Accessed: 07 April 2024).

Géron, A. (2019) Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems. O'REILLY.

Htait, A. (2021). 'BERT', CS985: Machine Learning for Data Analytics. Available at: https://classes.myplace.strath.ac.uk/mod/resource/view.php?id=1498815 (Accessed: 07 April 2024).