**MGMT 59000: Analyzing Unstructured Data**

**Team Assignment 3**

**Group 1**

**Part 1**

**Step 1**

data = pd.read\_excel('Assignment 3.xlsx')

# Building the training and test datasets

# Filtering the first 400 restaurant reviews and the first 400 movie reviews

train\_restaurant = data[(data['label'] == 'restaurant') & (data['id'] <= 400)]

train\_movie = data[(data['label'] == 'movie') & (data['id'] >= 501) & (data['id'] <= 900)]

# Combining the two datasets to form the training dataset

train\_dataset = pd.concat([train\_restaurant, train\_movie])

# The rest of the data will be used as the test dataset

test\_dataset = data.drop(train\_dataset.index)

# Checking the first few rows of each dataset to ensure they are correct

train\_dataset.head(), test\_dataset.head()

A screenshot of a computer program

Description automatically generated

**Step 2**

from sklearn.feature\_extraction.text import TfidfVectorizer

import nltk

from nltk.stem import WordNetLemmatizer

from nltk.corpus import stopwords

import string

# Define a custom tokenizer function

def custom\_tokenizer(text):

# Initialize WordNetLemmatizer

lemmatizer = WordNetLemmatizer()

# Split the text into words

words = text.split()

# Lemmatize, remove stop words and punctuations

tokens = [lemmatizer.lemmatize(word) for word in words

if word.lower() not in stopwords.words('english')

and word not in string.punctuation]

return tokens

# Initialize TfidfVectorizer

tfidf\_vectorizer = TfidfVectorizer(tokenizer=custom\_tokenizer, min\_df=5, ngram\_range=(1, 2))

# Apply TF-IDF transformation to the training dataset

tfidf\_matrix = tfidf\_vectorizer.fit\_transform(train\_dataset['review'])

# To view the shape of the TF-IDF matrix and feature names

print(tfidf\_matrix.shape)

print(tfidf\_vectorizer.get\_feature\_names\_out()[:10])

A screen shot of a computer

Description automatically generated

**Step 3**

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

from sklearn.naive\_bayes import MultinomialNB

from sklearn.linear\_model import LogisticRegression

from sklearn.ensemble import RandomForestClassifier

from sklearn.svm import SVC

from sklearn.neural\_network import MLPClassifier

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(tfidf\_matrix, train\_dataset['label'], test\_size=0.3, random\_state=42)

# Initialize models

naive\_bayes\_model = MultinomialNB()

logit\_model = LogisticRegression()

random\_forest\_model = RandomForestClassifier(n\_estimators=50)

svm\_model = SVC()

ann\_model = MLPClassifier(hidden\_layer\_sizes=(4,), max\_iter=1000)

# Dictionary to store models

models = {

"Naive Bayes": naive\_bayes\_model,

"Logistic Regression": logit\_model,

"Random Forest": random\_forest\_model,

"SVM": svm\_model,

"ANN": ann\_model

}

# Train each model and calculate accuracy

for name, model in models.items():

model.fit(X\_train, y\_train)

predictions = model.predict(X\_test)

accuracy = accuracy\_score(y\_test, predictions)

print(f"{name} Accuracy: {accuracy:.2f}")

A screenshot of a computer

Description automatically generated

Inference: Naïve Bayes model performs the best as per the accuracy rate. Logistic Regression follows it by being the second best with 0.99 accuracy.

But a point to consider might be that the Naive Bayes model reaching an accuracy of 1.00 could suggest overfitting, where the model has learned to perform perfectly on the training data but might not generalize well to unseen data.

**Step 4**

from keras.preprocessing.text import Tokenizer

from keras.preprocessing.sequence import pad\_sequences

# Number of words to consider as features

max\_features = 10000

# Maximum length of each document

maxlen = 100

# Initialize the tokenizer with a maximum number of words

tokenizer = Tokenizer(num\_words=max\_features)

# Fit the tokenizer on the training data

tokenizer.fit\_on\_texts(train\_dataset['review'])

# Convert the texts to sequences

train\_sequences = tokenizer.texts\_to\_sequences(train\_dataset['review'])

test\_sequences = tokenizer.texts\_to\_sequences(test\_dataset['review'])

# Pad the sequences so they all have the same length

X\_train = pad\_sequences(train\_sequences, maxlen=maxlen)

X\_test = pad\_sequences(test\_sequences, maxlen=maxlen)

# To view the shape of the sequences

print(X\_train.shape)

print(X\_test.shape)

A screenshot of a computer program

Description automatically generated

**Step 5**

from keras.models import Sequential

from keras.layers import Embedding, LSTM, Dropout, Dense

from keras.preprocessing.sequence import pad\_sequences

from keras.preprocessing.text import Tokenizer

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

import numpy as np

# Assuming max\_features and maxlen are defined as before

max\_features = 10000

maxlen = 100

# Initialize and fit the tokenizer

tokenizer = Tokenizer(num\_words=max\_features)

tokenizer.fit\_on\_texts(train\_dataset['review'])

# Convert texts to sequences

train\_sequences = tokenizer.texts\_to\_sequences(train\_dataset['review'])

test\_sequences = tokenizer.texts\_to\_sequences(test\_dataset['review'])

# Pad sequences

X\_train = pad\_sequences(train\_sequences, maxlen=maxlen)

X\_test = pad\_sequences(test\_sequences, maxlen=maxlen)

# Convert labels to numeric

y\_train = np.array(train\_dataset['label'].apply(lambda x: 1 if x == 'movie' else 0))

y\_test = np.array(test\_dataset['label'].apply(lambda x: 1 if x == 'movie' else 0))

# Define the model

model = Sequential()

model.add(Embedding(max\_features, 20, input\_length=maxlen))

model.add(LSTM(40, dropout=0.2, recurrent\_dropout=0.2))

model.add(Dropout(0.1))

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

# Compile the model

model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

# Train the model

model.fit(X\_train, y\_train, batch\_size=100, epochs=10, validation\_data=(X\_test, y\_test))

# Evaluate the model

loss, accuracy = model.evaluate(X\_test, y\_test)

print(f'Accuracy: {accuracy:.2f}')

A screenshot of a computer

Description automatically generated

Inference:

While comparing accuracy rates from the traditional machine learning models in step 3 with the deep learning model in step 5, we observe that the Naive Bayes, Logistic Regression, and SVM models all had higher accuracy rates than the deep learning model, with Naive Bayes even reaching perfect accuracy. The Random Forest and simple ANN models had accuracy rates that were equivalent to that of the deep learning model.

* While the traditional models showed higher accuracy rates on the training set; the deep learning model's performance is respectable and with further training, more data, and hyperparameter tuning, it could potentially outperform the traditional models. The LSTM's ability to understand sequence might make it a better choice for more complex classification tasks.
* The warning from the ANN model in step 3 suggests that the model had not converged, which might indicate that the deep learning model could outperform it with appropriate tuning.
* The traditional models might have performed exceptionally well due to the nature of the dataset. If the dataset is straightforward or if the distinctive features between restaurant and movie reviews are easily captured by simple statistical models, the additional complexity of a deep learning model might not offer a significant advantage.

**Part 2**

**Step 1**

from keras.datasets import cifar10

import matplotlib.pyplot as plt

# Load CIFAR-10 dataset

(x\_train, y\_train), (x\_test, y\_test) = cifar10.load\_data()

# Define the CIFAR-10 classes

classes = ['airplane', 'automobile', 'bird', 'cat', 'deer',

'dog', 'frog', 'horse', 'ship', 'truck']

# Visualize the first 20 images of the test set

plt.figure(figsize=(10, 4))

for i in range(20):

plt.subplot(2, 10, i+1)

plt.imshow(x\_test[i])

plt.title(classes[y\_test[i][0]])

plt.axis('off')

plt.tight\_layout()

plt.show()

A screenshot of a computer

Description automatically generated

The output shows the first 20 images of the CIFAR-10 test set along with their corresponding labels. The CIFAR-10 dataset consists of 60,000 32x32 color images in 10 different classes, with 6,000 images per class. The dataset is split into 50,000 training images and 10,000 test images.

The labels visible in the image are:

* Cat
* Ship
* Ship
* Airplane
* Frog
* Frog
* Automobile
* Frog
* Cat
* Automobile
* Airplane
* Truck
* Dog
* Horse
* Truck
* Ship
* Dog
* Horse
* Ship
* Frog

These labels correspond to the CIFAR-10 classes defined. The dataset includes a variety of objects that are commonly used for computer vision tasks such as image classification. The classes include modes of transportation like airplanes, trucks, and ships; animals like cats, dogs, frogs, horses, and birds; and other categories such as automobiles and deer.

The visualization demonstrates the diversity and complexity of the dataset, with images varying in scale, pose, and background. For instance, some images may show the object up close and centered, while others might include the object in a more complex environment or from different angles. This variability makes CIFAR-10 a challenging dataset for developing and testing image classification algorithms.

Having a balanced dataset with an equal number of images for each class helps to ensure that a trained model does not become biased toward more frequently represented classes. It also provides a good test bed for evaluating how well a model can generalize from training data to new, unseen images.

**Step 2**

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv2D, Dropout, MaxPooling2D, Flatten, Dense

from tensorflow.keras.utils import to\_categorical

from tensorflow.keras.datasets import cifar10

# Load CIFAR-10 dataset

(x\_train, y\_train), (x\_test, y\_test) = cifar10.load\_data()

# Normalize the data

x\_train = x\_train.astype('float32')

x\_test = x\_test.astype('float32')

x\_train /= 255

x\_test /= 255

# Convert class vectors to binary class matrices

y\_train = to\_categorical(y\_train, 10)

y\_test = to\_categorical(y\_test, 10)

# Define the CNN model

model = Sequential([

Conv2D(32, (3, 3), activation='relu', input\_shape=x\_train.shape[1:]), # a

Dropout(0.2), # b

Conv2D(32, (3, 3), activation='relu'), # c

MaxPooling2D(pool\_size=(2, 2)), # d

Conv2D(64, (3, 3), activation='relu'), # e

Dropout(0.2), # f

Conv2D(64, (3, 3), activation='relu'), # g

MaxPooling2D(pool\_size=(2, 2)), # h

Flatten(), # i

Dense(256, activation='relu'), # j

Dropout(0.2), # k

Dense(10, activation='softmax') # l

])

# Compile the model

model.compile(loss='categorical\_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train the model

batch\_size = 500 # Smaller than 500 to prevent overheating

epochs = 5

model.fit(x\_train, y\_train, batch\_size=batch\_size, epochs=epochs, validation\_data=(x\_test, y\_test), verbose=1)

# Evaluate the model

accuracy = model.evaluate(x\_test, y\_test, verbose=0)[1]

print(f'Accuracy: {accuracy \* 100:.2f}%')

A screenshot of a computer

Description automatically generated

Here's an interpretation of the results:

* Initial Low Accuracy: The model starts with an accuracy of 35.76% in the first epoch. This is expected as the model is just beginning to learn from the data.
* Steady Improvement: With each epoch, the model's accuracy improves, showing that it is learning from the training data. This is evidenced by the increase in accuracy from 35.76% in the first epoch to 61.56% by the fifth epoch.
* Learning Rate: The rate of improvement in accuracy appears to be diminishing with each epoch. This suggests that initial learning is rapid but begins to plateau as the model starts fitting the training data as much as it can with its given architecture.
* Validation Loss: The validation loss decreases with each epoch, which indicates that the model is getting better at generalizing to unseen data. The decrease from 1.7781 to 1.0774 suggests improvement in the model's ability to predict accurately on the test set.

From these results, we can infer that the CNN architecture with the specified layers and hyperparameters is capable of learning and improving its prediction accuracy over time. However, since the accuracy after 5 epochs is a bit over 60%, there is likely room for improvement either through further training, hyperparameter tuning, or architectural changes.

**Step 3**

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv2D, Dropout, MaxPooling2D, Flatten, Dense

from tensorflow.keras.utils import to\_categorical

from tensorflow.keras.datasets import cifar10

# Load CIFAR-10 dataset

(x\_train, y\_train), (x\_test, y\_test) = cifar10.load\_data()

# Normalize the data

x\_train = x\_train.astype('float32')

x\_test = x\_test.astype('float32')

x\_train /= 255

x\_test /= 255

# Convert class vectors to binary class matrices

y\_train = to\_categorical(y\_train, 10)

y\_test = to\_categorical(y\_test, 10)

# Define the modified CNN model

model\_modified = Sequential([

Conv2D(32, (3, 3), activation='relu', input\_shape=x\_train.shape[1:]), # a

Dropout(0.2), # b

Conv2D(32, (3, 3), activation='relu'), # c

MaxPooling2D(pool\_size=(2, 2)), # d

Conv2D(64, (3, 3), activation='relu'), # e

Dropout(0.2), # f

Conv2D(64, (3, 3), activation='relu'), # g

MaxPooling2D(pool\_size=(2, 2)), # h

Conv2D(128, (3, 3), activation='relu'), # step 3 a

Dropout(0.2), # step 3 b

Conv2D(128, (3, 3), activation='relu'), # step 3 c

Flatten(), # i

Dense(256, activation='relu'), # j

Dropout(0.2), # k

Dense(10, activation='softmax') # l

])

# Compile the modified model

model\_modified.compile(loss='categorical\_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train the modified model

batch\_size = 500 # Smaller than 500 to prevent overheating

epochs = 5

model\_modified.fit(x\_train, y\_train, batch\_size=batch\_size, epochs=epochs, validation\_data=(x\_test, y\_test), verbose=1)

# Evaluate the modified model

accuracy\_modified = model\_modified.evaluate(x\_test, y\_test, verbose=0)[1]

print(f'Accuracy: {accuracy\_modified \* 100:.2f}%')

A screenshot of a computer

Description automatically generated

Here's an analysis of these results:

* Starting Accuracy: The model begins with a similar accuracy to the previous model from step 2 at around 36%, which is expected since the model weights are initialized randomly.
* Accuracy Progression: The model's accuracy increases with each epoch, showing learning progress. However, the final accuracy after 5 epochs is 57.86%, which is lower than the 61.56% achieved in step 2.
* Comparing Model Performances: Despite the additional complexity added to the model in step 3, the accuracy has decreased compared to the model in step 2. This could be due to several factors, such as the model in step 3 potentially requiring more epochs to converge due to its increased complexity, or it could be beginning to overfit the training data.
* Validation Loss: The validation loss has not consistently decreased. It decreases initially but increases again at the 5th epoch, which might suggest that the model is not generalizing as well as the simpler model from step 2.

**Step 4**

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv2D, Dropout, MaxPooling2D, Flatten, Dense

from tensorflow.keras.utils import to\_categorical

from tensorflow.keras.datasets import cifar10

# Load and preprocess the CIFAR-10 dataset

(x\_train, y\_train), (x\_test, y\_test) = cifar10.load\_data()

x\_train = x\_train.astype('float32') / 255

x\_test = x\_test.astype('float32') / 255

y\_train = to\_categorical(y\_train, 10)

y\_test = to\_categorical(y\_test, 10)

# Define the modified CNN model (from step 3)

model\_modified = Sequential([

Conv2D(32, (3, 3), activation='relu', input\_shape=x\_train.shape[1:]), # a

Dropout(0.2), # b

Conv2D(32, (3, 3), activation='relu'), # c

MaxPooling2D(pool\_size=(2, 2)), # d

Conv2D(64, (3, 3), activation='relu'), # e

Dropout(0.2), # f

Conv2D(64, (3, 3), activation='relu'), # g

MaxPooling2D(pool\_size=(2, 2)), # h

Conv2D(128, (3, 3), activation='relu'), # step 3 a

Dropout(0.2), # step 3 b

Conv2D(128, (3, 3), activation='relu'), # step 3 c

Flatten(), # i

Dense(256, activation='relu'), # j

Dropout(0.2), # k

Dense(10, activation='softmax') # l

])

# Compile the model

model\_modified.compile(loss='categorical\_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train the model for 20 epochs

batch\_size = 500 # Smaller than 500 to prevent overheating

epochs\_extended = 20

model\_modified.fit(x\_train, y\_train, batch\_size=batch\_size, epochs=epochs\_extended, validation\_data=(x\_test, y\_test), verbose=1)

# Evaluate the model

accuracy\_extended = model\_modified.evaluate(x\_test, y\_test, verbose=0)[1]

print(f'Accuracy after 20 epochs: {accuracy\_extended \* 100:.2f}%')

A screenshot of a computer

Description automatically generated

A screenshot of a table

Description automatically generated

The results from step 4, which extend the training of the CNN from step 3 to 20 epochs, show a notable improvement in performance:

* Increased Accuracy: The model's accuracy has significantly increased to 74.28% after 20 epochs, compared to 57.86% after 5 epochs in step 3. This suggests that the additional layers added in step 3 needed more epochs to learn effectively from the data.
* Progress Over Time: The accuracy improves steadily over the 20 epochs, indicating that the model continues to learn and improve its predictions with more training. The plateauing of accuracy seen in the later epochs suggests that the model might be reaching the limits of what it can learn with its current configuration.
* Validation Loss: The validation loss decreases over time, which is a good sign that the model is generalizing well to unseen data. However, there are epochs where the validation loss slightly increases, which could be a sign of beginning to overfit, but it doesn't appear to be a major issue since the overall trend is downward.
* Comparing to Previous Results: The extended training has outperformed the previous 5-epoch trainings (61.56% in step 2 and 57.86% in step 3). This demonstrates the importance of sufficient training time, especially for more complex models.

From these results, the model benefits from additional training time. While more complex models can initially perform worse than simpler ones due to the requirement of more training to effectively tune the greater number of parameters, given enough epochs, they often outperform simpler models. The results of step 4 confirm this, as the model has now surpassed the performance of the model in step 2.

**Step 5**

from sklearn.metrics import accuracy\_score

from sklearn.naive\_bayes import MultinomialNB

from sklearn.ensemble import RandomForestClassifier

# Flatten the image data

x\_train\_flat = x\_train.reshape(x\_train.shape[0], -1)

x\_test\_flat = x\_test.reshape(x\_test.shape[0], -1)

# Reshape the labels for Naïve Bayes

y\_train\_flat = y\_train.argmax(axis=1)

y\_test\_flat = y\_test.argmax(axis=1)

# Train a Naïve Bayes model

naive\_bayes\_model = MultinomialNB()

naive\_bayes\_model.fit(x\_train\_flat, y\_train\_flat)

# Train a Random Forest model with 100 trees and max depth of 10

random\_forest\_model = RandomForestClassifier(n\_estimators=100, max\_depth=10, random\_state=42)

random\_forest\_model.fit(x\_train\_flat, y\_train\_flat)

# Predictions using Naïve Bayes

y\_pred\_nb = naive\_bayes\_model.predict(x\_test\_flat)

accuracy\_nb = accuracy\_score(y\_test\_flat, y\_pred\_nb)

# Predictions using Random Forest

y\_pred\_rf = random\_forest\_model.predict(x\_test\_flat)

accuracy\_rf = accuracy\_score(y\_test\_flat, y\_pred\_rf)

accuracy\_nb, accuracy\_rf

A screenshot of a computer program

Description automatically generated

The accuracy rates for the different models trained on the CIFAR-10 dataset are as follows:

* Step 2 (CNN - 5 Epochs): The CNN achieved an accuracy of 61.56%.
* Step 3 (Enhanced CNN - 5 Epochs): The more complex CNN with additional layers achieved an accuracy of 57.86%.
* Step 4 (Enhanced CNN - 20 Epochs): Extending the training to 20 epochs improved the accuracy to 74.28%.
* Step 5 (Naïve Bayes and Random Forest): The Naïve Bayes model yielded an accuracy of 29.33%, and the Random Forest model achieved 43.31%.

Comparing these results:

* CNN Models (Steps 2, 3, 4): CNNs performed the best, with the extended training CNN (Step 4) achieving the highest accuracy among all models. This is expected as CNNs are designed to work with image data by capturing spatial hierarchies and feature correlations.
* Traditional Machine Learning Models (Step 5): Both Naïve Bayes and Random Forest models performed significantly worse than CNNs. This is due to their inability to capture the complex patterns and spatial relationships in image data effectively.
* Best Performing Model: The model from Step 4, the enhanced CNN trained for 20 epochs, performed the best with an accuracy rate of 74.28%. The improvement in performance from steps 2 to 4 underscores the effectiveness of deeper architectures and longer training times for complex datasets like CIFAR-10.
* However, it's also important to note that while the CNN models outperform traditional machine learning models, they also tend to be more computationally intensive and may require more data and training time to achieve these results.