```
import pandas as pd
import numpy as np
import re
from nltk.stem import WordNetLemmatizer
from sklearn.model_selection import train_test_split
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Conv1D, MaxPooling1D,
Flatten, Dropout, Dense
from tensorflow.keras.regularizers import l2
from sklearn.metrics import confusion_matrix, classification_report,
roc_curve, auc
import matplotlib.pyplot as plt
import seaborn as sns
from imblearn.over_sampling import RandomOverSampler
from sklearn.utils.class_weight import compute_class_weight
import shap

# 1. Load and Preprocess the dataset
file_path = '/kaggle/input/requirment-csv/requirment.csv'  # Update this
to your Kaggle dataset path
df = pd.read_csv(file_path, encoding='latin1')
lemmatizer = WordNetLemmatizer()

def clean_text(text):
    text = text.lower()
    text = re.sub(r'[^\w\s]', '', text)
    text = re.sub(r'\d+', '', text)
    tokens = text.split()
    tokens = [lemmatizer.lemmatize(token) for token in tokens]
    return ' '.join(tokens)

df['Base_Reviews'] = df['Base_Reviews'].apply(clean_text)

# Tokenize and pad the sequences
tokenizer = Tokenizer(num_words=2000)  # Limit to max features
tokenizer.fit_on_texts(df['Base_Reviews'])
X = tokenizer.texts_to_sequences(df['Base_Reviews'])
vocab_size = len(tokenizer.word_index) + 1
maxlen = 100
X = pad_sequences(X, padding='post', maxlen=maxlen)

# Convert category labels to numerical values
y_dict = {'feature': 0, 'issue': 1, 'user_experience': 2,
'other_information': 3}
y = df['category'].map(y_dict)
y = pd.get_dummies(df['category']).values

# 2. Oversample to balance classes
oversampler = RandomOverSampler(random_state=42)
X_resampled, y_resampled = oversampler.fit_resample(X, y)
```

```
# Split data into train, validation, and test sets
X_train, X_temp, y_train, y_temp = train_test_split(X_resampled,
y_resampled, test_size=0.3, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp,
test_size=0.5, random_state=42)

# Calculate class weights
class_weights = compute_class_weight(class_weight='balanced',
classes=np.unique(np.argmax(y_train, axis=1)), y=np.argmax(y_train,
axis=1))
class_weights = dict(enumerate(class_weights))

# 3. Define the CNN model
def create_model():
    model = Sequential([
        Embedding(input_dim=vocab_size, output_dim=100,
input_length=maxlen),
        Conv1D(128, 5, activation='relu', kernel_regularizer=l2(0.01)),
        MaxPooling1D(pool_size=2),
        Flatten(),
        Dropout(0.2),  # Set dropout rate to 0.2
        Dense(64, activation='softmax', kernel_regularizer=l2(0.01))  #
Set dense layer to 64 units
    ])
    model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
loss='categorical_crossentropy', metrics=['accuracy'])
    return model

# Train the model
model = create_model()
history = model.fit(X_train, y_train, validation_data=(X_val, y_val),
epochs=10, batch_size=32, class_weight=class_weights, verbose=1)

# Print the keys in the history.history dictionary
print("Keys in history.history: ", history.history.keys())

# Plot learning curves
plt.figure(figsize=(10, 8))
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.savefig('bb_training_validation_accuracy.png', dpi=300)
plt.show()
plt.close()

# Confusion Matrix
y_pred = model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_true_classes = np.argmax(y_test, axis=1)
cm = confusion_matrix(y_true_classes, y_pred_classes)
plt.figure(figsize=(8, 6))
```

```
sns.heatmap(cm, annot=True, fmt='g')
plt.title('Confusion Matrix')
plt.savefig('bb_confusion_matrix.png', dpi=300)
plt.show()
plt.close()

# ROC Curve
n_classes = len(y_dict)
fpr, tpr, roc_auc = {}, {}, {}
for i in range(n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_test[:, i], y_pred[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

plt.figure(figsize=(10, 8))
colors = ['aqua', 'darkorange', 'cornflowerblue', 'red']
for i in range(n_classes):
    plt.plot(fpr[i], tpr[i], color=colors[i], lw=2, label=f'ROC curve for
{list(y_dict.keys())[i]} (area = {roc_auc[i]:.2f})')
plt.plot([0, 1], [0, 1], 'k--', lw=2)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='best')
plt.savefig('bb_roc_curve.png', dpi=300)
plt.show()
plt.close()

# Classification Report
print(classification_report(y_true_classes, y_pred_classes,
target_names=list(y_dict.keys())))

# SHAP explanations
distrib_samples = X_train[:100]

# SHAP KernelExplainer
explainer = shap.KernelExplainer(model.predict, distrib_samples)
num_explanations = 10
shap_values = explainer.shap_values(X_test[:num_explanations])

# Map word indices back to words
num2word = {v: k for k, v in tokenizer.word_index.items()}
x_test_words = np.stack([np.array(list(map(lambda x: num2word.get(x,
"NONE"), X_test[i]))) for i in range(num_explanations)])

# SHAP summary plot for all classes
plt.figure()
shap.summary_plot(shap_values, feature_names=[num2word.get(i, "NONE") for
i in range(vocab_size)], class_names=list(y_dict.keys()), show=False)
plt.savefig('bb_shap_summary_plot.png', dpi=300)
plt.close()

# SHAP summary plot for each class
```

```
for class_name in y_dict.keys():
    plt.figure()
    class_index = y_dict[class_name]
    shap.summary_plot(shap_values[class_index],
feature_names=[num2word.get(i, "NONE") for i in range(vocab_size)],
show=False)
    plt.title(f'SHAP Summary Plot for Class "{class_name}"')
    plt.savefig(f'bb_shap_{class_name}_summary_plot.png', dpi=300)
    plt.close()

# Define the class name and input number for SHAP force plot
class_name = 'feature'  # Change to the desired class name
input_num = 1  # Change to the desired input number

# Get the class number from the class name
class_num = y_dict[class_name]

# SHAP force plot for the specified class and input
force_plot = shap.force_plot(explainer.expected_value[class_num],
shap_values[class_num][input_num], x_test_words[input_num])
shap.save_html('bb_shap_force_plot.html', force_plot)

# SHAP force plot for the specified class across multiple inputs
multi_force_plot = shap.force_plot(explainer.expected_value[class_num],
shap_values[class_num], x_test_words[:num_explanations])
shap.save_html('bb_multi_shap_force_plot.html', multi_force_plot)
```