Configuration Manual for Crime Category Classification

Jerin Eldho x21196737

1 Introduction

This configuration manual provides a comprehensive guide for setting up and implementing a research project focused on accurately categorizing crime incidents based on textual descriptions. Various machine learning models, including LSTM, GRU, Logistic Regression, SVM, and Random Forest, are employed to achieve this objective.

2 Environment Setup

Device Specifications:

- 1. Model Name: MI Notebook Pro
- 2. Processor: Intel Core i5-11300H @ 3.10GHz (Base) up to 3.11GHz (Turbo Boost), 11th Gen
- 3. Memory (RAM): 16.0 GB DDR4
- 4. Graphics: Integrated Graphics (Intel Xe Graphics)
- 5. Operating System: Windows 11 Home Single Language (64-bit)

Required Software:

- 1. Development Environment: Google Colab
- 2. Programming Language: Python 3.8.16
- 3. Cloud Storage Integration: Google Drive

These software components are essential for the proper functioning of the project. Google Colab provides the computational environment for executing code, while Python 3.8.16 serves as the programming language for implementing the project's algorithms and data processing. Integration with Google Drive enables seamless data access and storage, facilitating efficient collaboration and data management throughout the research endeavor.

3 Python Libraries Essential for Implementing this Project

The below figure shows the python libraries used for implementing this project:



Figure 1: Importing python libraries

4 Loading the dataset

To access the dataset stored in Google Drive, use the following code snippet to mount Google Drive within the Colab environment and read the CSV data into a DataFrame named "crimedata":



[] crime_data = pd.read_csv(google_drive_path)

Figure 2: Loading dataset

5 Data Preprocessing and Cleaning

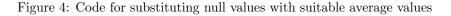
The dataset was initially downloaded from the SFPD website, containing records from 2018 to 2023. To ensure data integrity:

Data from 2022 to 2023 was extracted (Figure 3). Missing values were handled by substituting them with appropriate average values (Figure 4). The "Incidentdatetime" column was converted into a datetime format for precise temporal referencing (Figure 5). The "IncidentDayofWeek" column was transformed into numerical categorical values for machine learning compatibility (Figure 6). Categories were mapped to a new unified category for improved modeling (Figure 7).



Figure 3: Extracting data from 2022 till 2023





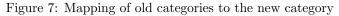
	# Converting "Incident_datetime" column to datetime format
	date formats = ['%2-%+%r', '%/%m/%d %2:%2-%5 %p']
	for date format in date formats:
	try:
	filtered of['Incident Datatime'] = pd.to_datatime(Filtered_df['Incident_Datatime'], format=date_format)
	break
	except Valueirror:
	9855 C
	# Filtering records only for 2021-2023
	start_date = pd.to_datetime('2021-00-01')
	end_date = pd.to_datetime('2023-12-31')
	crime_data1 = filtered_df[(filtered_df['Incident_Datetime'] >= start_date) & (filtered_df['Incident_Datetime'] <= end_date)]
	# Printing the DataFrame after preprocessing
	prist(crime_data1.shape(0))

Figure 5: Conversion of Incidentdatetime column into a datetime format



Figure 6: Conversion of IncidentDayofWeek column into numerical categorical values





6 Feature Engineering and Visualization

The code snippet in Figure 8 demonstrates feature extraction from the "Incidentdatetime" column, including date, year, month, day, hour, minute, hour type, season, and weekend feature. Various plots were generated for data visualization:

Extracting date,year,month,day,hour,minute etc. from 'Incident_Datetime' column
def extract_datetime_features(df):
<pre>df['Incident_Datetime'] = pd.to_datetime(df['Incident_Datetime']) # Convert 'Incident_date' column to pandas datetime</pre>
Extracting year, month, date, day, hour, and minute
df['Year'] = df['Incident_Datetime'].dt.year
df['Month'] = df['Incident_Datetime'].dt.month
df['Day'] = df['Incident_Datetime'].dt.day
df['Hour'] = df['Incident_Datetime'].dt.hour
<pre>df['Hinste'] = df['Incident_Datetime'].dt.minute</pre>
Extracting hour type (morning, afternoon, evening, night)
df['Hour type'] = df['Incident Datetime'].dt.hour.apply(lambda x: 'Morning' if 6 <= x < 12 else
"Afternoon" if 12 <= x < 18 else
'Evening' if 18 <= x < 22 else 'Wight')
Extracting season (winter, summer, fall, spring)
df['Season'] = df['Incident Datetime'].dt.month.apply(lambda m: 'Winter' if m in [12, 1, 2] else
'Spring' if m in [3, 4, 5] else
'Summer' if m in [6, 7, 8] else 'Fall')
Adding weekend feature (1 for weekend, 0 for weekdays)
<pre>df['Weekend'] = df['Incident_Datatime'].dt.dayofweek.apply(lambda x: 1 if x >= 5 else 0)</pre>
return df
Callng the function to extract datetime features and update the DataFrame
crime_data1 = extract_datetime_features(crime_data1)

Figure 8: Code for extracting different features from 'Incidentdatetime' column

The following figures display the code snippets for plotting various distributions, crime categories distribution, distribution of crimes by year and police district, and occurrences of crimes on a yearly basis:

```
[ ] # Defining a function of plotting various distributions
  def plot_distribution(data, column_name):
        plt.figure(figsize=(10, 6))
        sns.countplot(x=column_name, data=data)
        plt.title(f'Distribution of {column_name}')
        plt.title(f'Distribution of {column_name}')
        plt.ylabel(column_name)
        plt.ylabel('Count')
        plt.show()
[ ] plot_distribution(crime_df, 'Police_District')
        plt.show()
[ ] plot_distribution(crime_df, 'Police_District')
        plot_distribution(crime_df, 'Year')
        plot_distribution(crime_df, 'Month')
        plot_distribution(crime_df, 'Season')
        plot_distribution(crime_df, 'Weekend')
```

Figure 9: Code for plotting various distributions

[] # Plotting Crime Categories Distribution plt.figure(figsize=(12, 6)) sns.countplot(x='Unified_category', data=crime_df) plt.title('Crime Categories Distribution') plt.xlabel('Unified Category') plt.ylabel('Count') plt.xticks(rotation=90) plt.show()

Figure 10: Code for plotting Crime Categories Distribution

[] # Distribution of Crimes by Year and Police District plt.figure(figsize=(12, 6)) sns.countplot(x='Police_District', hue='Year', data=crime_df) plt.title('Distribution of Crimes by Year and Police District') plt.xlabel('Police District') plt.xlabel('Count') plt.legend(title='Year', loc='upper right') plt.xticks(rotation=45) plt.show()

Figure 11: Code for plotting Distribution of Crimes by Year and Police District

Ploting the occurrences of crimes that had happened yearly basis. plt.figure(figsize=(12, 6)) sns.countplot(x='Year', data=crime_df, hue='Unified_category', palette='tab10') plt.title('Occurrences of Crimes by Year and Crime Category') plt.xlabel('Year') plt.ylabel('Count') plt.legend(title='Crime Category', bbox_to_anchor=(1, 1)) plt.xticks(rotation=45) plt.show()

Figure 12: Code for ploting the occurrences of crimes that had happened yearly basis

7 Implementation of various machine learning models

The implementation of machine learning models, including LSTM, GRU, Logistic Regression, SVM, and Random Forest, is illustrated in the following figures:

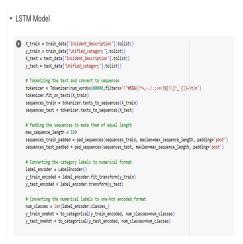


Figure 13: Text Data Preprocessing and Label Encoding of LSTM Model

D	# Creating the LSTN model
	embedding_dim = 125
	lstm_units = 64
	model = Sequential()
	model.add(Embedding(input_dim=len(tokenizer.word_index) + 1, output_dim=embedding_dim, input_length=max_sequence_leng
	model.add(LSTM(units=256,activation='tanh',return_sequences=True))
	model.add(LSTH(unitsel28,activations'tanh'))
	model.add(Dense(num_classes, activations'softmax'))
	# Compiling the model
	<pre>model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])</pre>
	# Training the model
	batch_size = 64
	epochs = 5
	model.fit(sequences_train_padded, y_train_onehot, batch_sizesbatch_size, epochs.epochs, validation_splitu0.1)
	# Evaluating the model
	loss, accuracy = model.evaluate(sequences_test_padded, y_test_onehot)
	print(f"Test loss: (loss), Test accuracy: (accuracy)")
	# Generate predictions
	y_pred_onehot = model.predict(sequences_test_padded)
	# Convert predictions to labels
	y_pred_encoded = np.argmax(y_pred_omehot, axis=1)
	y_pred_labels = label_encoder.inverse_transform(y_pred_encoded)
	# Convert true labels to labels
	y_true_encoded = np.argmax(y_test_onehot, axiss1)
	<pre>y_true_labels = label_encoder.inverse_transform(y_true_encoded)</pre>
	# Calculate confusion matrix
	<pre>confusion_mtx = confusion_matrix(y_true_labels, y_pred_labels, labels=label_encoder.classes_)</pre>
	# Print confusion matrix
	print("Confusion Matrix:")
	print(confusion_mtx)

Figure 14: LSTM Model





Figure 15: Text Data Preprocessing and Label Encoding of GRU Model

0	s Crating the 600 sode) embedding dis = 1100 gruguitt = 64
	<pre>modl : Sequential() modl.add(bencing(upt,disele(tsenier.asr(jdse) + 1, ortpd_disededing_dis_ipt_legthese_sequenc_legth)) modl.add(bencing(upt,disele(tsenier.asr(jdsenier_lyne modl.add(bencing), equent_1, stutients=thm', return_sequence(heat, kard(_reglaris=210.01))) modl.add(bencing(upt,disele, stutient=thm', return_sequence(heat, kard(_reglaris=210.01))) modl.add(bencing(upt,dise, stutient=thm', return=1))</pre>
	<pre># Compiling the model model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])</pre>
	# Training the model
	betch size = 64
	erochs = 5
	<pre>nodel.fit(sequences_train_padded, y_train_onehot, batch_size=batch_size, epochs=epochs, validation_split=0.1)</pre>
	# Evaluating the model
	loss, accuracy = model.evaluate(sequences test padded, v test onehot)
	<pre>print(f"Test loss: (loss), Test accuracy: (accuracy)")</pre>
	# Generate predictions
	y_pred_onehot = model.predict(sequences_test_padded)
	# Convert predictions to labels
	<pre>y_pred_encoded = np.argmax(y_pred_onehot, axis=1)</pre>
	<pre>y_pred_labels = label_encoder.inverse_transform(y_pred_encoded)</pre>
	# Convert true labels to labels
	<pre>y_true_encoded = np.argmax(y_test_onehot, axis=1)</pre>
	<pre>y_true_labels = label_encoder.inverse_transform(y_true_encoded)</pre>
	# Calculate confusion matrix
	confusion_mtx = confusion_matrix(y_true_labels, y_pred_labels, labels+label_encoder.classes_)
	# Print confusion matrix
	print("Confusion Matrix:")
	print(confusion_wtx)

Figure 16: GRU Model

✓ Logistic Regression Model

0	X_trein = train_data('Incldent_Description').tolist() y_trein = train_data('Unified_category').tolist() X_text = test_data('Incldent_Description').tolist() y_text = test_data('Unified_ategory').tolist()
	<pre># Step 1: Converting the category labels to numerical format label_encoder = Label_encoder() y_train_encode = label_encoder.fit_transform(y_train) y_test_encoded = label_encoder.transform(y_test)</pre>
	<pre># Step 2: Vectorizing the text data using TfidfVactorizer vectorizer = TfidfVectorizer(stop_words:emplish, max_features=5000) X_train_vectors = vectorizer(t_transform(X_train) X_test_vectors = vectorizer.transform(X_test)</pre>
	# Step 3: Creating and training the Logistic Regression model logreg_model = LogisticRegression(max_iter=1000) logreg_model.fit(X_train_vectors, y_train_encoded)
	<pre># Step 4: Making predictions on the test data y_pred = logreg_model.predict(X_test_vectors)</pre>
	<pre># Step 5: Converting numerical predictions back to category labels y_pred_labels = label_encoder.inverse_transform(y_pred)</pre>
	<pre># Step 6: Evaluating the model accuracy = accuracy_score(y_test, y_pred_labels) print("Accuracy:", accuracy)</pre>

Classification report for detailed performance metrics
print(classification_report(y_test, y_pred_labels))

Figure 17: Logistic Regression Model

		+ Cod
[]	X train = train data['Incident Description'].tolist()	
	v train = train data ('Unified category'), tolist()	
	X test = test date(incident Description 1.tolist()	
	<pre>y_test = test_data['Unified_category'].tolist()</pre>	
	# Step 1: Converting the category labels to numerical format	
	label_encoder = LabelEncoder()	
	<pre>y_train_encoded = label_encoder.fit_transform(y_train)</pre>	
	<pre>y_test_encoded = label_encoder.transform(y_test)</pre>	
	# Step 2: Vectorizing the text data using TfidfVectorizer	
	vectorizer = TfidfVectorizer(stop_words='english', max_features=5000)	
	X_train_vectors = vectorizer.fit_transform(X_train)	
	X_test_vectors = vectorizer.transform(X_test)	
	# Step 3: Creating and training the Support Vector Machine (SVM) model	
	sym_model = SVC(kernel='linear') # You can experiment with different kernels (linear, rbf, etc.)	
	<pre>svm_model.fit(X_train_vectors, y_train_encoded)</pre>	
	# Step 4: Making predictions on the test data	
	<pre>y_pred = svm_model.predict(X_test_vectors)</pre>	
	# Step 5: Converting numerical predictions back to category labels	
	<pre>y_pred_labels = label_encoder.inverse_transform(y_pred)</pre>	
	# Step 6: Evaluating the model	
	accuracy = accuracy_score(y_test, y_pred_labels)	
	print("Accuracy:", accuracy)	
	#Classification report for more detailed performance metrics	
	print(classification report(v test, v pred labels))	

Figure 18: SVM model

→ Random Forest	
<pre>[]</pre>	
<pre>X_train_vectors = vectorizer.fit_transform(X_train) X_test_vectors = vectorizer.transform(X_test) # Step 3: Creating and training the Random Forest model random_forest_model = RandomForestClassifiar(m_stimators100, random_state random_forest_model.fit(X_train_vectors, y_train_mcoded)</pre>	=42
<pre># Step 4: Making predictions on the test data y_pred = random_forest_model.predict(X_test_vectors) # Step 5: Converting numerical predictions back to category labels</pre>	
<pre># step 5: Lonverting numerical predictions dack to category labels y_pred_labels = label_encoder.inverse_transform(y_pred) # Step 6: Evaluating the model</pre>	
<pre># step 6: Evaluating the model accuracy = accuracy.test, y_pred_lebels) print("Accuracy:", accuracy)</pre>	
<pre># classification report for more detailed performance metrics print(classification_report(y_test, y_pred_labels))</pre>	

Figure 19: Random Forest