**Machine Learning Project: Crime Type Prediction**

**1. Problem Definition and Goal**

This project aims to predict the type of crime committed based on various features in Los Angeles crime data. By predicting crime types accurately, authorities and organizations can allocate resources efficiently and gain valuable insights into crime patterns. The ultimate goal is to develop a classifier model that can predict the crime type (crm\_cd\_desc) based on other features such as location, weapon used, and victim demographics.

* **Objective**: Build a classifier that predicts the type of crime.
* **Target Variable**: crm\_cd\_desc (crime type)

**2. Data Overview**

The data used for this project is sourced from Google BigQuery's dataset on Los Angeles crime data. This dataset contains various features, including:

* **Crime description** (crm\_cd\_desc): The target variable we aim to predict.
* **Location** (location), **weapon description** (weapon\_desc), **premise description** (premis\_desc), etc.
* **Victim details**: Age (vict\_age), descent (vict\_descent), gender (vict\_sex).
* **Date and time details**: These help in understanding crime trends over time, but are not used directly in this version of the code.

**3. Data Preprocessing**

The first crucial step in any machine learning project is data preprocessing. Here's what was done:

* **Dropping unnecessary columns**: The dr\_no, date\_occ, and date\_rptd columns were dropped as they don't contribute significantly to the prediction.
* **Removing duplicates**: The dataset was cleaned by dropping any duplicate entries.

python

Copy

df = df.drop(columns=['dr\_no', 'date\_occ', 'date\_rptd'])

df = df.drop\_duplicates()

* **Feature Engineering**:
  + **Frequency Encoding**: High-cardinality categorical columns like mocodes, location, and cross\_street were frequency encoded. This helps convert these variables into a more manageable form by using their relative frequencies as values.

python

Copy

high\_cardinality\_columns = ['mocodes', 'location', 'cross\_street']

for col in high\_cardinality\_columns:

frequency\_encoding = df[col].value\_counts(normalize=True)

df[f'{col}\_encoded'] = df[col].map(frequency\_encoding)

df[f'{col}\_encoded'] = df[f'{col}\_encoded'].fillna(0)

* + **Label Encoding**: Low-cardinality categorical columns like premis\_desc and weapon\_desc were label encoded, transforming them into numerical values for use in machine learning models.

python

Copy

low\_cardinality\_columns = ['premis\_desc', 'weapon\_desc']

label\_encoder = LabelEncoder()

for col in low\_cardinality\_columns:

df[f'{col}\_encoded'] = label\_encoder.fit\_transform(df[col])

* + **One-Hot Encoding**: Columns such as area\_name, vict\_descent, and status\_desc were one-hot encoded to represent categorical variables as binary vectors.

python

Copy

df = pd.get\_dummies(df, columns=['area\_name', 'vict\_descent', 'status\_desc'])

* **Handling Missing Values**:
  + vict\_age (victim's age) was missing for some entries. The missing values were filled with the median age of the victims.

python

Copy

X\_train['vict\_age'] = X\_train['vict\_age'].fillna(X\_train['vict\_age'].median())

* **Splitting the Data**:
  + The dataset was split into training and testing sets (80% for training, 20% for testing). Additionally, missing values in the crm\_cd\_\* columns were replaced with zeros to handle any empty entries during model training.

python

Copy

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

**4. Model Selection and Justification**

For the model selection, two different classifiers were considered: **Logistic Regression** and **Random Forest**.

* **Logistic Regression** was used first as a baseline. Logistic regression works well when the data is linearly separable, but it may not perform as well when the data has complex, non-linear relationships.

python

Copy

model = LogisticRegression(max\_iter=1000, verbose=1)

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

y\_pred = model.predict(X\_test)

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

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

* + **Advantages** of using Logistic Regression:
    - Simplicity: Easy to interpret.
    - Fast to train.
  + **Limitations**:
    - Assumes linearity, which might not work well for complex datasets.
* **Random Forest** would likely be considered in the next steps due to its capability to model non-linear relationships and its robustness to overfitting.

**5. Model Training and Evaluation**

* **Training the Model**:
  + The Logistic Regression model was trained using the training set (X\_train, y\_train). Training time was tracked using the time module to measure efficiency.

python

Copy

start\_time = time.time()

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

end\_time = time.time()

print(f"Total time taken: {end\_time - start\_time:.2f} seconds")

* **Evaluation**:
  + The model was evaluated on the test set (X\_test) and accuracy was calculated. For classification tasks, additional evaluation metrics such as precision, recall, and F1-score should be considered, especially in the case of imbalanced classes.

**6. Results and Interpretation**

* **Model Performance**:
  + Couldn’t run the model because of the huge dataset. Tried Google Colab as well but that also couldn’t provide necessary resources for this level of intensive computation within its free tier limits
* **Key Observations**: – observation based ChatGPT explanation.
  + Some classes might be underrepresented, leading to potential misclassifications.
  + The model performs well on certain crime types but may struggle with rare or less frequent types.
* **Feature Importance**:
  + While Logistic Regression doesn’t inherently provide feature importance, models like **Random Forest** or **Gradient Boosting** could be used in the future to gain insights into which features are driving the predictions.

**7. Next Steps and Improvements**

1. **Model Comparison**:
   * **Random Forest** might work better for this type of problem due to its ability to handle complex data and interactions.
2. **Hyperparameter Tuning**: Use this for model improvement in future.

**8. Reflection and Learning**

This project was an excellent introduction to the process of data preprocessing, feature engineering, and applying machine learning algorithms. Key learnings include:

* Understanding how to handle missing data, feature encoding, and dimensionality reduction.
* Learning about the limitations of different machine learning algorithms and how to optimize them.

**Version 2 of the Crime Prediction Project**

**Changes in the Second Version (Detailed with Output)**

In the second version of the project, several optimizations and improvements were implemented to address system resource challenges and enhance model performance. Below are the key updates along with the corresponding output:

**1. Handling Large Dataset and System Resources**

* **Change**: In the first version, the model was trained on the entire dataset, which caused issues with system memory and performance. To address this, the dataset was limited to **50,000 rows** to avoid overwhelming the system's resources. This decision was made after encountering severe system health issues when processing the full dataset.

python

Copy

query = """

SELECT \* FROM `la-crime-analysis.la\_crime\_dataset.crime\_data`

LIMIT 50000 # Limiting the dataset to 50,000 rows to avoid memory overload

"""

* **System Health Monitoring**: The system's health was monitored using psutil to keep track of CPU, memory, and disk usage at key stages of the process (before and after querying the data, preprocessing, and model training). This allowed for better resource management throughout the project.
* **Output Before Query Execution**:

pgsql

Copy

CPU Usage: 7.0%

Memory Usage: 9.20 GB / 15.79 GB

Disk Usage: 261.45 GB / 324.71 GB

----------------------------------------

* **Output After Loading Data**:

pgsql

Copy

Time taken to load data: 5.76 seconds

CPU Usage: 12.4%

Memory Usage: 9.25 GB / 15.79 GB

Disk Usage: 261.45 GB / 324.71 GB

----------------------------------------

* **Reasoning**: By reducing the dataset size, the data processing and model training steps were performed more efficiently, without overloading the system's memory.

**2. Data Preprocessing Optimizations**

* **Efficient Encoding for High-Cardinality Columns**: Frequency encoding was applied to high-cardinality columns like mocodes, location, and cross\_street. After encoding, the original columns were dropped to save memory and reduce dimensionality.

python

Copy

high\_cardinality\_columns = ['mocodes', 'location', 'cross\_street']

for col in high\_cardinality\_columns:

if col in df.columns:

frequency\_encoding = df[col].value\_counts(normalize=True)

df[f'{col}\_encoded'] = df[col].map(frequency\_encoding).fillna(0)

df = df.drop(columns=[col])

* **Output After Preprocessing**:

yaml

Copy

System Health After Preprocessing:

CPU Usage: 13.0%

Memory Usage: 9.22 GB / 15.79 GB

Disk Usage: 261.45 GB / 324.71 GB

----------------------------------------

Post-processing memory usage: 11.84 MB

* **Memory Optimization**: Data types were optimized to reduce memory usage. The vict\_age column was converted to numeric, and the crm\_cd\_desc column was cast to category, which significantly reduced the memory footprint.

python

Copy

df['vict\_age'] = pd.to\_numeric(df['vict\_age'], errors='coerce').fillna(df['vict\_age'].median())

df['crm\_cd\_desc'] = df['crm\_cd\_desc'].astype('category')

* **Imputation of Missing Values**: Missing values in the feature set were handled using the SimpleImputer with a median imputation strategy, ensuring that no rows were dropped, which is crucial for maintaining the dataset's size.

python

Copy

imputer = SimpleImputer(strategy='median') # Use median for numerical columns

X = pd.DataFrame(imputer.fit\_transform(X), columns=X.columns)

**3. Model Training and Evaluation**

* **Logistic Regression Model**: A **Logistic Regression** model was used with the liblinear solver, as it performs well with smaller datasets and binary classification tasks. The model training time was recorded to monitor the efficiency of the training process.

python

Copy

model = LogisticRegression(max\_iter=1000, solver='liblinear')

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

* **Output for Model Training**:

css

Copy

Time taken to train Logistic Regression: 811.59 seconds

* **Accuracy**: After training the model, predictions were made on the test set, and the model's accuracy was evaluated using accuracy\_score. The accuracy achieved was **64%**.

python

Copy

y\_pred = model.predict(X\_test)

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

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

**Output**:

makefile

Copy

Accuracy: 0.64

* **Reasoning**: The accuracy of **0.64** indicates the model's capability in making predictions. While not perfect, this provides a reasonable starting point, and improvements can be made through model optimization techniques like hyperparameter tuning and feature engineering.

**4. Performance and Resource Monitoring**

* **System Health After Model Training**:

yaml

Copy

System Health After Model Training:

CPU Usage: 8.8%

Memory Usage: 9.20 GB / 15.79 GB

Disk Usage: 261.44 GB / 324.71 GB

----------------------------------------

* **Total Runtime**: The total runtime for the process, from data extraction to model training, was recorded as **823.12 seconds**.

python

Copy

total\_runtime = time.time() - total\_start\_time

print(f"Total Runtime: {total\_runtime:.2f} seconds")

**Output**:

yaml

Copy

Total Runtime: 823.12 seconds

**5. Key Takeaways and Future Work**

1. **Model Tuning**: Further steps could include hyperparameter tuning using techniques such as **GridSearchCV** or **RandomizedSearchCV** to improve the model's performance.
2. **Scalability**: If system resources allow, the dataset size could be increased for potentially better model performance. Additionally, larger models like **Random Forest** or **XGBoost** could be explored for more complex patterns in the data.
3. **Class Imbalance**: Future improvements should consider addressing any class imbalances within the target variable (crm\_cd\_desc). Techniques like **SMOTE** (Synthetic Minority Over-sampling Technique) or **class weighting** could improve performance on underrepresented classes.
4. **System Monitoring**: The system health monitoring approach proved useful for ensuring that resource usage was kept in check, and it should continue to be used as the dataset grows in size.

**Conclusion**

The second version of the crime prediction project introduces key optimizations to handle system resource limitations effectively. By limiting the dataset size and optimizing preprocessing steps, the project now runs efficiently with the available system resources. The performance of the Logistic Regression model, while modest at 64% accuracy, sets a strong foundation for further improvements and model exploration.

**Version 3 - Enhancements and Modifications**

**Changes from Previous Versions:**

1. **Memory Management and Efficiency:**
   * In this version, the focus was on optimizing memory usage by applying multiple techniques such as:
     + **Variance Threshold** for feature selection, which removes features that have constant values (i.e., those that do not vary across samples).
     + **StandardScaler** was applied for feature scaling, ensuring that all features have the same scale for better model performance.
     + **Principal Component Analysis (PCA)** was used to reduce the dimensionality of the dataset from the full set of features to 20 principal components. This reduces both the number of features and potential overfitting by retaining the most important information.
2. **Data Preprocessing:**
   * **Encoding:** As in previous versions, categorical variables were handled using **LabelEncoder** for low-cardinality columns and **Frequency Encoding** for high-cardinality columns.
   * **Memory Optimization:** The **crm\_cd\_desc** target column was optimized to a categorical type to reduce memory usage.
   * **Handling Missing Values:** Missing values were imputed using the **SimpleImputer** with the strategy set to 'median' to avoid dropping rows.
3. **System Monitoring:**
   * A function to monitor **system health** was added throughout the script, allowing you to keep track of the **CPU usage**, **memory usage**, and **disk usage** at different stages of the execution. This was useful in understanding how the system resources were being utilized at various points and helped identify any potential issues during model training.
4. **Modeling:**
   * **Random Forest Classifier** was chosen for classification due to its effectiveness in handling large datasets with complex features. The model was trained and evaluated on the processed data.
   * **Evaluation Metrics:** The model's performance was assessed using accuracy and a classification report, which included precision, recall, and F1-score for each class.

**System Health and Model Execution Details:**

**System Health Monitoring (Before, During, After Execution):**

* **Before Query Execution:**
  + CPU Usage: 23.1%
  + Memory Usage: 9.12 GB / 15.79 GB
  + Disk Usage: 261.45 GB / 324.71 GB
  + This shows that the system is operating at moderate levels before any data manipulation begins.
* **After Data Loading:**
  + CPU Usage: 7.8%
  + Memory Usage: 9.22 GB / 15.79 GB
  + Disk Usage: 261.45 GB / 324.71 GB
  + Memory usage slightly increased after loading data but remained within acceptable limits.
* **After Data Preprocessing:**
  + CPU Usage: 3.6%
  + Memory Usage: 9.21 GB / 15.79 GB
  + Disk Usage: 261.45 GB / 324.71 GB
  + The system shows minimal resource usage post-preprocessing, indicating the code's efficiency in reducing the dataset's memory footprint.
* **Before Model Training:**
  + CPU Usage: 9.1%
  + Memory Usage: 9.24 GB / 15.79 GB
  + Disk Usage: 261.45 GB / 324.71 GB
  + Memory and CPU usage are still low before model training, suggesting the system is ready for heavy computations.
* **After Model Training:**
  + CPU Usage: 13.0%
  + Memory Usage: 11.03 GB / 15.79 GB
  + Disk Usage: 261.45 GB / 324.71 GB
  + After the model training, CPU usage and memory usage increased due to the processing power needed for the Random Forest algorithm.

**Output Metrics and Model Evaluation:**

* **Training Time:**
  + The time taken to train the Random Forest model was **38.35 seconds**, which was relatively fast given the complexity of the data and model.
* **Accuracy:**
  + The accuracy of the model was **0.63**. Although this is a reasonable baseline, there is potential to improve this with further feature engineering, hyperparameter tuning, or by exploring more advanced models.
* **Class Distribution:**
  + A key observation was the **class imbalance** in the target variable crm\_cd\_desc. Certain classes like 'VEHICLE - STOLEN' and 'BATTERY - SIMPLE ASSAULT' have significantly higher counts compared to others. This imbalance could have contributed to the model's performance in terms of accuracy, as it is likely biased towards the more frequent classes.
* **Classification Report (Sampled):**
  + Below is a snippet from the classification report:

pgsql

Copy

precision recall f1-score support

ARSON 0.78 0.28 0.41 25

ASSAULT WITH DEADLY WEAPON ON POLICE OFFICER 0.00 0.00 0.00 11

WEAPONS POSSESSION/BOMBING 0.00 0.00 0.00 1

accuracy 0.63 10000

macro avg 0.28 0.19 0.20 10000

weighted avg 0.60 0.63 0.60 10000

* + **Precision** for most classes is low, and this is likely due to the class imbalance, as the model has a tendency to favor the more frequent classes.
  + The **weighted average** precision, recall, and F1-score are relatively decent, suggesting that the model is performing better on the majority class.

**Total Runtime:**

* The total runtime of the entire process was **51.44 seconds**, which includes loading the data, preprocessing, training the model, and evaluating it. The total time spent is manageable, considering the size of the dataset and the complexity of the operations performed.

**Conclusion:**

This version represents significant progress in terms of model efficiency and system resource management. Key improvements included reducing dimensionality, scaling features, and optimizing memory usage. Although the model's performance was decent, further improvements in feature engineering, hyperparameter tuning, and addressing class imbalance could potentially increase its accuracy and robustness. This version also demonstrates the ability to handle larger datasets (up to 50,000 rows) while monitoring system health, making it a good foundation for future iterations.