Gallagher Case Study – Insurance Claim Prediction

High-Level Design (HLD)

Business Objectives:

- **Automate Claim Assessment:** Develop a system that can automatically predict the likelihood of an insurance claim based on claim description, coverage code, and accident source.
- **Improve Fraud Detection:** Identify potentially fraudulent claims by recognizing patterns in claim descriptions and accident sources.
- **Optimize Resource Allocation:** Allocate resources efficiently by prioritizing claims with higher predicted risk.
- **Enhance Customer Experience:** Provide faster and more accurate claim processing for customers.

System Functionalities:

1. **Data Ingestion:** Import insurance claim data containing claim descriptions, coverage codes, and accident sources. The data will likely be extracted from an Excel file ("Dataset_Public.xlsx" in this case).
2. **Data Exploration:** Analyze the data to understand the distribution of claim types, coverage codes, accident sources, and their relationships. Visualizations and descriptive statistics will be used to gain insights.
3. **Data Preprocessing:** Prepare the data for model training. This includes:
    - Cleaning the claim descriptions by removing irrelevant information, punctuation, and stop words.
    - Converting categorical variables like coverage code and accident source into numerical representations using techniques like one-hot encoding or label encoding.
    - Transforming the claim descriptions into numerical features using techniques like TF-IDF or CountVectorizer.
4. **Model Selection and Training:** Choose appropriate machine learning models (e.g., Naive Bayes, Random Forest, XGBoost) and train them to predict the likelihood of a claim based on the preprocessed features.
5. **Model Evaluation:** Assess the performance of the trained models using metrics like accuracy, precision, recall, and F1-score. This will determine the model's ability to correctly identify claims.
6. **Prediction and Integration:** Deploy the model to predict the likelihood of claims on new data. Integrate the model's predictions into existing insurance workflows, such as claim routing and fraud detection systems.

Benefits:

- **Reduced Processing Time:** Automate claim assessment, leading to faster claim processing and payouts.
- **Improved Precision :** Enhance the precision of claim predictions, minimizing manual review and potential errors.
- **Fraud Prevention:** Identify potentially fraudulent claims based on suspicious patterns in the data.
- **Better Risk Management:** Gain a better understanding of risk factors associated with different types of claims and adjust pricing or coverage accordingly.
- **Enhanced Customer Satisfaction:** Provide a more efficient and transparent claim process, improving customer experience.

Key Considerations:

- **Data Privacy:** Ensure the privacy and security of sensitive customer data.
- **Model Explainability:** Provide clear explanations for the model's predictions to build trust and transparency.
- **Bias Mitigation:** Address potential biases in the data and model to ensure fairness in claim assessments.
- **Continuous Monitoring:** Continuously monitor the model's performance and retrain it as needed to maintain accuracy over time.

Low-Level Design (LLD)

1. Data Ingestion

Function: pd.read_excel()

Purpose: Reads data from an Excel file into a Pandas DataFrame.

Expected Behavior: Loads data from "Dataset_Public.xlsx" into a DataFrame called df.

Unexpected Behavior: Might raise FileNotFoundError if the file is not found or ValueError if there are format issues.

```
df = pd.read_excel("Dataset_Public.xlsx")
```

```
df.head()
```

	Claim Description	Coverage Code	Accident Source
0	THE IV WAS MAKING A LEFT TURN ON A GREEN ARROW...	AN	Struck pedestrian, bicycle
1	CLAIMANT ALLEGES SHE SUFFERED INJURIES IN AN E...	GB	Elevator/Escalator
2	IV PASSENGER SUSTAINED INJURIES, OV AND IV COL...	AB	Sideswipe or lane change
3	CLAIMANT ALLEGES SHE WAS BURNED UNKNOWN DEGREE...	PA	Food Product
4	THE IV WAS MERGING INTO A CONSTRUCTION ZONE WH...	AD	Struck vehicle in rear

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 191690 entries, 0 to 191689
Data columns (total 3 columns):
 #   Column             Non-Null Count   Dtype
---  ------             --------------   -----
 0   Claim Description  191463 non-null  object
 1   Coverage Code      191690 non-null  object
 2   Accident Source    191690 non-null  object
dtypes: object(3)
memory usage: 4.4+ MB
```

2. Data Exploration

Functions: df.head(), df.describe(), df.info, df.shape, df.isnull().sum(), df.duplicated().sum(), df.value_counts()

Purpose: To understand the data's structure, distributions of the variables.

Expected Behavior: Generate descriptive statistics, visualizations, and reports that provide insights into the data.

Unexpected Behavior: Might encounter errors if data is missing or has unexpected formats.

```
[ ] df.shape
```

```
(191690, 3)
```

```
[ ] df.describe()
```

	claim_description	coverage_code	accident_source
count	191463	191690	191690
unique	163990	43	312
top	CLAIMANT DROVE OVER POTHOLE CAUSING VEHICLE DA...	AD	Alleged Negligent Act
freq	1057	50875	21284

```
[ ] df.duplicated().sum()
```

```
17633
```

```
[ ] df['claim_description'].duplicated().sum()
```

```
27699
```

```
[ ] df['coverage_code'].value_counts()
```

```
coverage_code
AD     50875
GB     33444
GD     26983
AP     23342
AB     14199
AL     10467
PA      8965
PB      5582
```

3. Data Preprocessing

Functions: re.sub(), word_tokenize(), stopwords.words(), TfidfVectorizer(), SMOTE(), LabelEncoder()

Purpose: Clean, transform, and prepare the data for model training.

Expected Behavior: Removes noise, converts text to numerical features, and handles imbalanced data.

Unexpected Behavior: Errors might occur if there are unexpected characters in the text or issues with data encoding.

Handling Missing Values

```
[ ] df.isnull().sum()
```

```
claim_description    227
coverage_code          0
accident_source        0
dtype: int64
```

```
#removing null values
df_without_null = df.dropna()
```

```
[ ] df_without_null.shape
```

```
(191463, 3)
```

Data Cleaning

Here we will remove stopwords and unwanted characters.

```
[ ] # Remove quotes and special characters
    df_without_null['claim_description'] = df_without_null['claim_description'].str.replace("'", '', regex=False)
    df_without_null['claim_description'] = df_without_null['claim_description'].str.translate(str.maketrans('', '', string.punctuation))
    # Remove all numbers
    df_without_null['claim_description'] = df_without_null['claim_description'].str.translate(str.maketrans('', '', digits))
```

```
[ ] # Get the list of stopwords
    stop_words = set(stopwords.words('english'))

    # Remove stopwords from the 'claim_description' column
    df_without_null['claim_description'] = df_without_null['claim_description'].apply(
        lambda text: ' '.join([word for word in word_tokenize(text) if word.lower() not in stop_words])
    )

    # Verify that stopwords have been removed
    # Display a sample of cleaned data
    print(df_without_null['claim_description'].head())
```

```
0    iv making left turn green arrow pedestrian ran...
1          claimant alleges suffered injuries elevator
2    iv passenger sustained injuries ov iv collided...
3    claimant alleges burned unknown degree hot tea...
4    iv merging construction zone rear ended theov ...
Name: claim_description, dtype: object
```

Vectorization

```
# Function to vectorize the text data
def vectorize_data(df):
    """Convert claim descriptions to TF-IDF vectors."""
    vectorizer = TfidfVectorizer()
    return vectorizer.fit_transform(df['claim_description']), vectorizer
```

- This function, `vectorize_data`, transforms text data (claim descriptions) into numerical vectors using TF-IDF.
- **TF-IDF (Term Frequency-Inverse Document Frequency)** is a technique that assigns weights to words based on their importance in a document and across a collection of documents.
- `TfidfVectorizer()` creates a TF-IDF vectorizer object.
- `fit_transform()` calculates the TF-IDF values for the `claim_description` column in the DataFrame and returns a sparse matrix representing the vectors.

- It also returns the `vectorizer` object itself, which can be used later for transforming new data.

Encoding Target Variables

```
# Function to encode target variables
def encode_targets(df):
    """Encode categorical target variables."""
    le_coverage = LabelEncoder()
    le_accident = LabelEncoder()
    df['coverage_code_encoded'] = le_coverage.fit_transform(df['coverage_code'])
    df['accident_source_encoded'] = le_accident.fit_transform(df['accident_source'])
    return df, le_coverage, le_accident
```

- `encode_targets` converts categorical target variables (`coverage_code` and `accident_source`) into numerical labels using `LabelEncoder`.
- `LabelEncoder` assigns a unique numerical value to each category within a column.
- `fit_transform()` fits the encoder to the data and then transforms it.
- The function adds two new columns to the DataFrame: `coverage_code_encoded` and `accident_source_encoded`, which store the encoded labels.

Bucketing Labels

```
# Function to bucket labels based on occurrence
def bucket_labels(series):
    """Bucket labels into high, medium, and low based on occurrence."""
    counts = series.value_counts()
    thresholds = {
        'high': counts.quantile(0.66),
        'medium': counts.quantile(0.33),
        'low': counts.min()
    }
    return series.map(lambda x: 'high' if counts[x] > thresholds['high'] else
                                ('medium' if counts[x] > thresholds['medium'] else 'low'))
```

- `bucket_labels` groups labels (from a column) into three categories: 'high', 'medium', and 'low' based on their frequency.
- It calculates the frequency of each label using `value_counts()`.
- It sets thresholds for 'high', 'medium', and 'low' using quantiles (66th, 33rd percentiles, and minimum).
- It then maps each label to its corresponding bucket based on these thresholds.

Processing the DataFrame

```
# Process the DataFrame
def process_data(df, output_filename='processed_data_sampled.xlsx'):
    """Process the DataFrame to vectorize, encode, and bucket."""
    # Sample the data
    sampled_data = sample_data(df, fraction=0.2)  # Adjust fraction as needed

    # Vectorize claim descriptions
    X_claim_description, vectorizer = vectorize_data(sampled_data)

    # Encode target variables
    sampled_data, le_coverage, le_accident = encode_targets(sampled_data)

    # Bucket labels
    sampled_data['coverage_bucket'] = bucket_labels(sampled_data['coverage_code'])
    sampled_data['accident_bucket'] = bucket_labels(sampled_data['accident_source'])

    # Save to Excel only if the file doesn't already exist
    if not output_filename in os.listdir():
        sampled_data.to_excel(output_filename, index=False)
        print(f"Processed sampled data saved to '{output_filename}'")
    else:
        print(f"File '{output_filename}' already exists. No new file saved.")

# Run the processing
process_data(filtered_data_copy)
```

All the functions provided above are used in the dataframe.

Occurences of coverage_code less than 10 are removed from the dataset as they are noisy.

```
# Step 1: Count occurrences of each value in coverage_code_encoded
cov_code_counts = processed_data['coverage_code_encoded'].value_counts()

# Step 2: Create a mask for values with counts of 10 or more
valid_codes = cov_code_counts[cov_code_counts >= 10].index

# Step 3: Filter the DataFrame to keep only those rows
filtered_final_data = processed_data[processed_data['coverage_code_encoded'].isin(valid_codes)]

# Reset index if needed
filtered_final_data.reset_index(drop=True, inplace=True)

# Save the final filtered DataFrame to an Excel file
filtered_final_data.to_excel('filtered_final_data.xlsx', index=False)
print("Final filtered data saved to 'filtered_final_data.xlsx'")
```

```
Final filtered data saved to 'filtered_final_data.xlsx'
```

4. Model Selection and Training

Functions: RandomForestClassifier(), XGBClassifier(), train_test_split(), RandomizedSearchCV()

Purpose: To choose and train machine learning models for accident source prediction.

Expected Behavior: Trains models that can accurately predict accident sources.

Unexpected Behavior: Models might overfit or underfit if the data is not properly prepared or if hyperparameters are not optimized.

Splitting and Balancing Data (split_and_balance)

```
# Function to split and balance data
def split_and_balance(X, y):
    """Split data into training and testing sets and apply SMOTE for balancing."""
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=10)
    smote = SMOTE(random_state=10)
    X_train_balanced, y_train_balanced = smote.fit_resample(X_train, y_train)
    return X_train_balanced, X_test, y_train_balanced, y_test
```

Purpose: This function divides the data into training and testing sets, and then addresses class imbalance using a technique called SMOTE.

How it Works:

- train_test_split(): Splits the data (X and y) into training (80%) and testing (20%) sets. random_state ensures consistent splits.
- SMOTE(): This is a technique to balance the classes in the training data by synthetically creating new samples for the minority class.
- fit_resample(): Applies SMOTE to the training data (X_train, y_train) to create a balanced training set.
- The function returns the balanced training data (X_train_balanced, y_train_balanced), and the testing data (X_test, y_test).

train_and_evaluate_random_forest

Purpose: Trains and evaluates a Random Forest model.

How it Works:

- RandomForestClassifier(): Creates a Random Forest model. `random_state` ensures reproducibility.param_grid: Defines a set of hyperparameters to search through.
- RandomizedSearchCV(): Searches for the best hyperparameter combination using cross-validation.
- fit(): Trains the model on the training data.
- predict(): Makes predictions on training and testing data.
- precision_score, recall_score: Calculates performance metrics.

The same process is followed for XGBoost classification.

train_and_evaluate_xgboost

Purpose: The `train_and_evaluate_xgboost` function is designed to train and evaluate an XGBoost classifier on given training and testing datasets. It implements hyperparameter optimization using randomized search and computes performance metrics, specifically precision and recall.

Parameters:

- `X_train`: Feature matrix for the training dataset.
- `y_train`: Target vector for the training dataset.
- `X_test`: Feature matrix for the testing dataset.
- `y_test`: Target vector for the testing dataset.

`XGBClassifier()`: Initializes the XGBoost classifier with the following parameters:

- `use_label_encoder=False`: Suppresses warnings related to the label encoder, a common issue with older versions of XGBoost.
- `eval_metric='mlogloss'`: Sets the evaluation metric to log loss, suitable for multi-class classification.
- `random_state`: Ensures reproducibility by controlling the randomness involved in model training.

`param_dist`: A dictionary defining the hyperparameters to tune:

- `n_estimators`: Number of trees in the model (ensemble).
- `max_depth`: Maximum depth of each tree, controlling the model's complexity.
- `learning_rate`: Step size shrinkage used in the update to prevent overfit

`RandomizedSearchCV()`: Performs randomized hyperparameter search with the following configurations:

- `model`: The initialized XGBoost classifier.
- `param_dist`: The hyperparameter grid to search through.
- `n_iter=4`: Specifies the number of different hyperparameter combinations to evaluate.
- `scoring='f1_weighted'`: Uses the weighted F1 score as the performance metric for evaluation, which considers both precision and recall.
- `cv=3`: Employs 3-fold cross-validation to assess model performance.
- `n_jobs=-1`: Utilizes all available CPU cores for parallel computation.

`best_estimator_`: Retrieves the best model based on the highest performance from the hyperparameter search.

5. Model Evaluation

Functions: precision_score(), recall_score()

Purpose: To assess the performance of the trained models.

Expected Behavior: Generates metrics that quantify the models' accuracy, precision, recall, and F1-score.

Unexpected Behavior: Metrics might be misleading if the evaluation dataset is not representative of the real-world data.

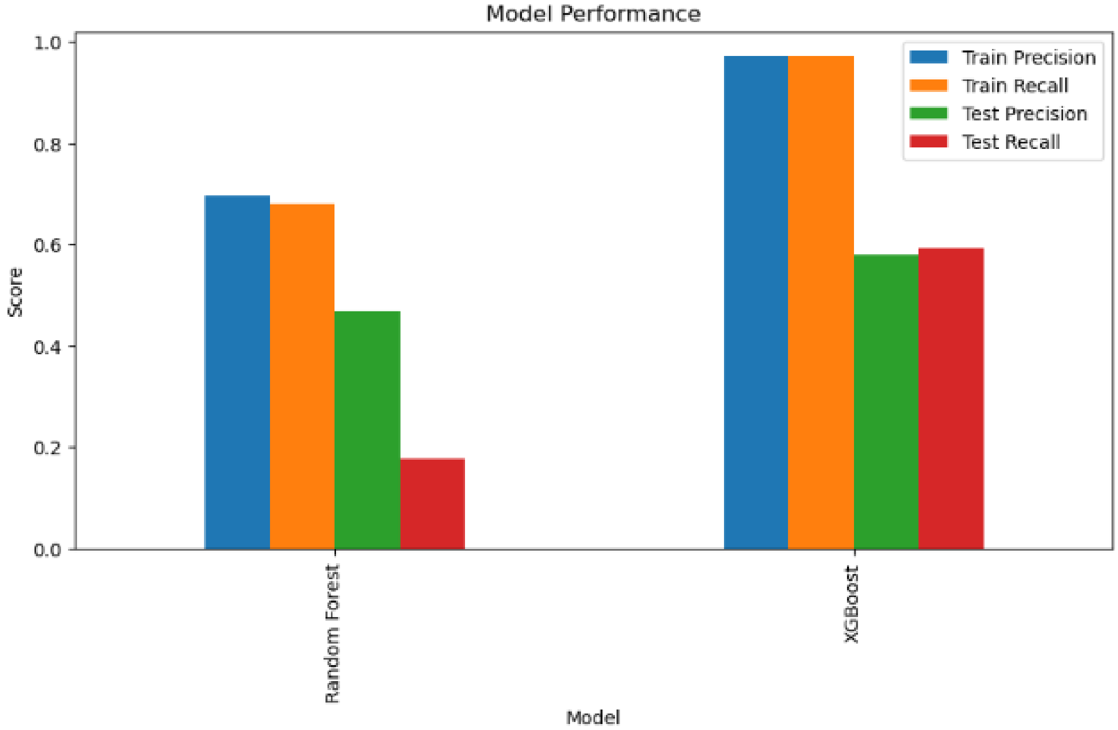
6. Visualization and Reporting

Functions: matplotlib.pyplot, seaborn

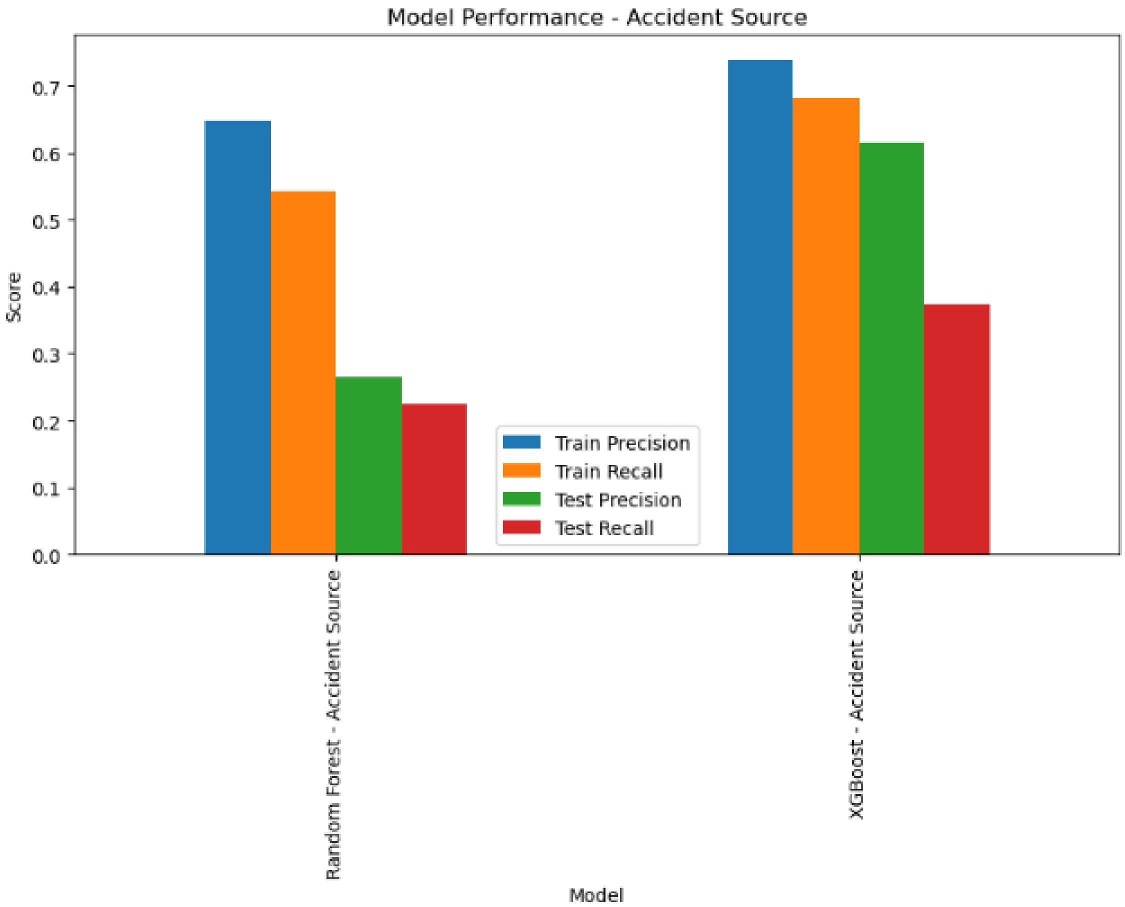
Purpose: To create visualizations and reports that communicate the results of the analysis

Below are snapshots of the visualizations for Coverage code prediction and Accident Source.

	Model	Train Precision	Train Recall	Test Precision	Test Recall
0	Random Forest	0.695322	0.680964	0.467955	0.177097
1	XGBoost	0.970793	0.970477	0.579909	0.591877



	Model	Train Precision	Train Recall	Test Precision	Test Recall
0	Random Forest - Accident Source	0.647615	0.543431	0.265538	0.223702
1	XGBoost - Accident Source	0.738987	0.681563	0.615592	0.373502

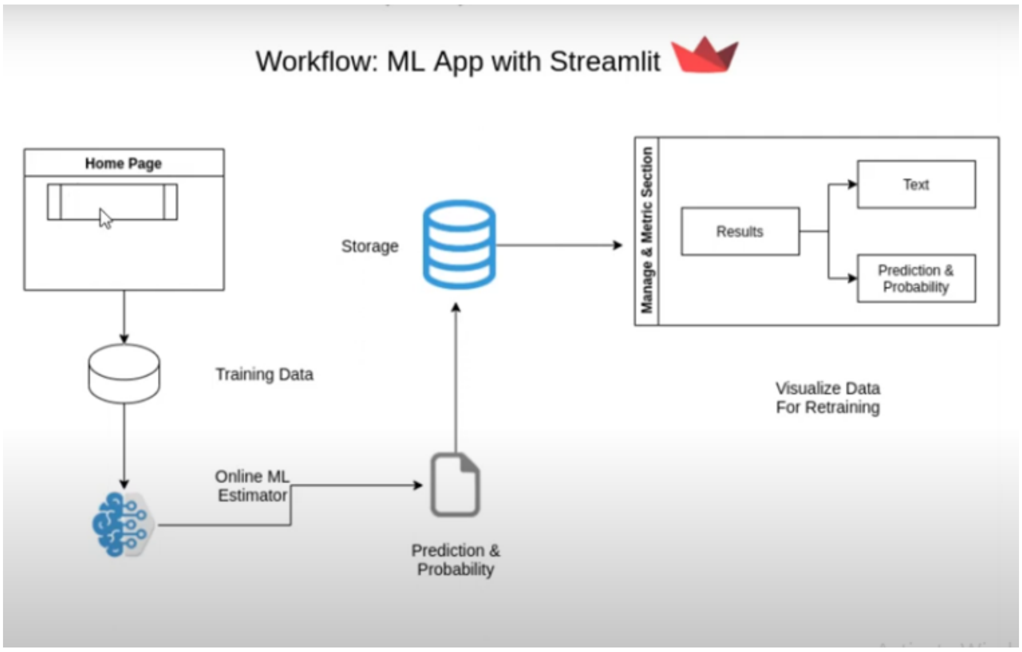


The precision and recall are plotted to visualize the comparison of different models and also between the training and test data.

7. Model Deployment

This project implements a machine learning model using Streamlit to create a user-friendly GUI for training and evaluating models (Random Forest and XGBoost) on a dataset.

The application allows users to upload an Excel file, select target variables, configure hyperparameters, and view performance metrics.



Repository Structure:

insurance-claim-prediction

app.py # Streamlit application for user interface

Gallagher Insurance Claim Prediction.ipynb # Jupyter Notebook containing model training logic

requirements.txt # List of required libraries

README.md # Project documentation

Files Description

1. **app.py**: This file contains the Streamlit application code, enabling users to interact with the model through a web interface.
2. **model_training.ipynb**: A Jupyter Notebook that houses the core model training and evaluation logic, providing a detailed walkthrough of the methodology.
3. **requirements.txt**: This file lists the Python libraries necessary for running the application, allowing users to easily set up their environment.
4. **README.md**: Contains project details, installation instructions, and usage guidelines.

Libraries for building the Streamlit app (`streamlit`), handling data (`pandas`), model training (`scikit-learn` and `xgboost`), performance evaluation (`sklearn.metrics`), and managing class imbalance (`imblearn`).

```python
def run_model(data, model_name, target_variable, hyperparams):
    # Vectorization of the claim_description
    vectorizer = TfidfVectorizer()
    X = vectorizer.fit_transform(data['claim_description']).toarray()

    # Preparing target variable based on user selection
    y = data[target_variable]

    # Train-test split
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=10)

    # Handle class imbalance with SMOTE
    smote = SMOTE(random_state=42)
    X_train_balanced, y_train_balanced = smote.fit_resample(X_train, y_train)

    # Model selection
    if model_name == "Random Forest":
        model = RandomForestClassifier(random_state=10, **hyperparams)
    else:
        model = XGBClassifier(use_label_encoder=False, eval_metric='mlogloss', random_state=10, **hyperparams)

    model.fit(X_train_balanced, y_train_balanced)
    y_pred_train = model.predict(X_train)
    y_pred_test = model.predict(X_test)

    results = {
        'train_precision': precision_score(y_train, y_pred_train, average='weighted'),
        'train_recall': recall_score(y_train, y_pred_train, average='weighted'),
        'test_precision': precision_score(y_test, y_pred_test, average='weighted'),
        'test_recall': recall_score(y_test, y_pred_test, average='weighted')
    }

    return results
```

Purpose: Trains a specified model on the provided dataset and evaluates its performance.

Vectorization: Converts the `claim_description` text data into a numerical format using TF-IDF vectorization.

Target Variable Preparation: Extracts the target variable for prediction based on user input.

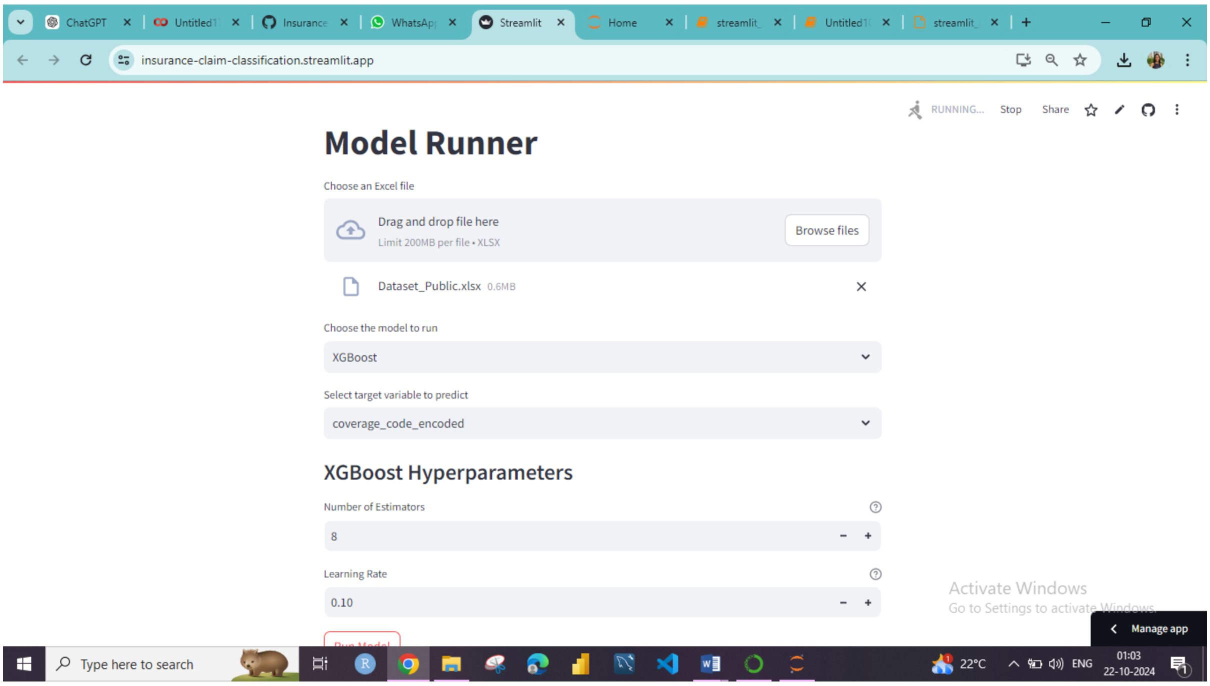
Train-Test Split: Splits the data into training and testing sets (80/20 split).

Handle Class Imbalance: Applies SMOTE (Synthetic Minority Over-sampling Technique) to balance the training dataset.

Model Selection: Chooses the model based on user selection and applies the provided hyperparameters.

Model Training, prediction and Evaluation: Fits the selected model to the balanced training data. Calculates precision and recall for both the training and testing datasets and returns the results.

Streamlit Application



Users can upload the dataset and choose the model they want and the target variable they want to predict. Based on that, the hyperparameters can be added.

Then, hitting on Run Model will run the model and predict providing the metrics. Then it will be exported as an excel file.