Machine Learning Project Documentation

Project Title: Classifying Socioeconomic Vulnerability

Team Members

1. **Dugo Gadisa**
2. **Galata Waqwaya**
3. **Wabi Jifara**

May 2025

Table of Contents

Model Refinement Documentation

Overview

Model refinement is a critical phase in the machine learning process that focuses on optimizing the performance of a previously trained model. The initial model, trained using the training data, may exhibit shortcomings such as underfitting (poor performance on both training and test data) or overfitting (excellent performance on training and poor performance on test data). Model refinement aims to address these issues and improve the model's ability to make accurate predictions on new, unseen data.

The key aspects of model refinement include:

- **Hyperparameter Tuning:** Machine learning models have parameters that are learned from the data and hyperparameters that are set by the data scientist *before* training. Hyperparameter tuning involves systematically searching for the optimal combination of these hyperparameter values to maximize model performance.
- **Feature Engineering:** This involves creating new features, modifying existing ones, or removing irrelevant ones to provide the model with more informative input data.
- **Model Selection:** In some cases, refining the current model may not be sufficient, and it may be necessary to switch to a different type of model altogether.
- **Regularization:** Techniques like L1 or L2 regularization can be used to prevent overfitting by adding a penalty term to the model's loss function.
- **Ensemble Methods:** Combining multiple models (e.g., bagging, boosting) can often lead to improved performance and robustness.

The goal of model refinement is to find the sweet spot where the model generalizes well to new data, exhibiting both accuracy and robustness. This iterative process requires careful experimentation, evaluation, and analysis of model performance metrics.

Model Evaluation

The initial model evaluation results show the performance of the machine learning models before any refinement. Key metrics to consider include accuracy, classification report (precision, recall, F1-score), and the confusion matrix. These metrics provide insight into how well the model is performing overall and in each class.

Areas for improvement may include:

- **Low Accuracy:** If the accuracy is low, it indicates that the model is not making correct predictions frequently. This could be due to underfitting, where the model is too simple to capture the underlying patterns in the data.
- **Class Imbalance:** The classification report and confusion matrix can reveal if the model performs poorly on certain classes. This is common in imbalanced datasets, where some classes have significantly fewer samples than others. For example, the model may have high precision but low recall for a minority class, indicating that it correctly identifies only a small fraction of instances of that class.
- **High Variance:** If the model performs much better on the training data than on the test data, it may be overfitting. This means the model is memorizing the training data rather than generalizing to new data.
- **Confusion Matrix:** The confusion matrix visualizes the model's predictions and can help identify specific types of errors. For example, it can show if the model is frequently confusing two particular classes.

Based on these initial results, model refinement techniques such as hyperparameter tuning, feature engineering, model selection, regularization, or ensemble methods can be applied to address these areas of improvement and enhance the model's predictive performance.

Refinement Techniques

The following techniques were used for refining the model:

- **Model Selection**: Four different classification algorithms were evaluated:
    - Logistic Regression
    - Decision Tree
    - Random Forest
    - XGBoost

The rationale is to compare the performance of different algorithms on the given dataset and select the one that generalizes best.

- **Data Preprocessing**:
    - Scaling: The numerical features were scaled using StandardScaler. This is important because some algorithms are sensitive to the magnitude of the features.
    - Encoding: The target variable (Vulnerability_Level) was encoded using LabelEncoder. This is necessary because the machine learning algorithms require numerical input.

By applying these refinement techniques, the goal is to improve the accuracy and generalization performance of the model.

Hyperparameter Tuning

In this project, hyperparameter tuning was primarily focused on the XGBoost model, as it generally offers strong performance. The following hyperparameters were considered:

- n_estimators: This parameter controls the number of boosting rounds. A higher number can improve performance but also increase the risk of overfitting.
- max_depth: The maximum depth of a tree. Deeper trees can capture more complex relationships but are also more prone to overfitting.
- learning_rate: This parameter scales the contribution of each tree. Lower values can improve generalization but require more training rounds.
- subsample: The fraction of samples used for fitting the individual trees. Lower values can reduce overfitting.
- colsample_bytree: The fraction of features used for fitting the individual trees.
- min_child_weight: Minimum sum of instance weight (hessian) needed in a child.

The tuning process involved a grid search approach, where different combinations of these hyperparameters were evaluated using cross-validation. The goal was to find the combination that yielded the best balance between bias and variance, resulting in improved generalization performance.

Insights and Impact:

- Initial experimentation showed that the default hyperparameters for XGBoost resulted in a model that slightly overfit the training data.
- Reducing the max_depth from its default value and lowering the learning_rate helped to mitigate this overfitting, leading to a more robust model.
- Adjusting the subsample and colsample_bytree parameters also contributed to a slight improvement in performance, as it introduced more randomness into the model building process, which can help to prevent overfitting.

The hyperparameter tuning process resulted in a noticeable improvement in the XGBoost model's performance, as evidenced by a higher accuracy score and improved precision and recall for all classes. The confusion matrix also indicated fewer misclassifications, demonstrating the effectiveness of the tuning process.

Cross-Validation

In this project, cross-validation was employed to ensure the robustness and generalization ability of the models, particularly during the hyperparameter tuning phase.

- **Initial Strategy**: The initial strategy involved using 5-fold cross-validation. This means the training data was divided into 5 subsets (folds). The model was trained on 4 folds and evaluated on the remaining fold, and this process was repeated 5 times, with each fold serving as the validation set once. This provided a reliable estimate of the model's performance.
- **Changes During Refinement**: No significant changes were made to the cross-validation strategy during the model refinement phase. The 5-fold cross-validation approach was maintained because it provided a good balance between computational cost and accuracy in estimating model performance.
- **Reasoning**:
    - Using cross-validation helps to prevent overfitting by providing a more robust estimate of the model's performance than a single train-test split.
    - 5-fold cross-validation is a common choice, offering a reasonable trade-off between the number of folds (which impacts computational time) and the variance of the performance estimate.
    - Maintaining the same cross-validation strategy throughout the refinement process ensures that the model comparisons and hyperparameter tuning are performed on a consistent basis.

Feature Selection

In this project, feature selection was not explicitly performed as a separate step during the model refinement phase. However, the initial data exploration and preprocessing steps implicitly addressed feature relevance.

- **Initial Data Exploration**: The initial data exploration phase, which involved analyzing the correlation matrix and feature distributions, provided insights into the relevance of each feature to the target variable (Vulnerability_Level). This analysis helped in understanding which features were most likely to be predictive of vulnerability.
- **Data Preprocessing**: During data preprocessing, some features might have been dropped if they were deemed irrelevant or redundant. For instance, if two features were highly correlated, one of them might have been dropped to avoid multicollinearity issues.
- **Model-Based Feature Importance**: Some of the models used, such as Random Forest and XGBoost, have built-in mechanisms for evaluating feature importance. These models can provide insights into which features are most influential in predicting the target

variable. While this information was not used to explicitly select a subset of features, it was used to understand the data better and validate the initial feature choices.

Explanation of Effects on Model Performance:

Since explicit feature selection methods were not the primary focus of the refinement process, the impact on model performance is indirect. However, the emphasis on data exploration and preprocessing ensured that the models were trained on the most relevant and informative features available. This likely contributed to improved model performance by:

- Reducing noise and redundancy in the data.
- Improving the model's ability to generalize to unseen data.
- Potentially simplifying the model and reducing the risk of overfitting.

Test Submission

Overview

The test submission phase involves preparing the refined model for final evaluation or deployment. Here's an overview of the typical steps:

a. **Final Model Selection**: Based on the results of the model refinement process, the best-performing model is selected. This selection is based on the evaluation metrics obtained during cross-validation and any other relevant considerations (e.g., computational cost, interpretability). In this case, XGBoost was chosen as the final model due to its superior performance.
b. **Retraining on Full Training Data**: The selected model (XGBoost) is retrained using the entire training dataset (X_train). This is crucial because the model was previously evaluated using cross-validation, where only a portion of the training data was used for training in each fold. Retraining on the full dataset maximizes the amount of information available to the model, potentially leading to further performance improvements.
c. **Preparation of Test Data**: The test dataset (X_test) is prepared in the same way as the training data. This includes any necessary preprocessing steps, such as scaling and encoding. It's essential to apply the same transformations to the test data as were applied to the training data to ensure consistency.
d. **Prediction on Test Data**: The retrained model is used to predict the target variable (Vulnerability_Level) for the test data (X_test). These predictions represent the model's final output.

e. **Evaluation of Test Set Performance**: The model's predictions on the test data are evaluated using the same metrics used during model refinement (e.g., accuracy, classification report, confusion matrix). This evaluation provides an unbiased estimate of the model's generalization performance on unseen data.

Data Preparation for Testing

The test dataset (X_test) is prepared in the same way as the training data. This involves applying the same preprocessing steps to ensure consistency and compatibility with the trained model.

Specific Considerations:

f. **Scaling**: If the training data was scaled, the test data must be scaled using the *same* scaling parameters (e.g., mean and standard deviation) derived from the training data. This prevents data leakage, where information from the test set influences the training process.

```
# Scale the test data using the same scaler fitted on the training data
X_test_scaled = scaler.transform(X_test)
```

* **Explanation**: Scaling is a crucial preprocessing step, especially for algorithms sensitive to the magnitude of features (e.g., Support Vector Machines, Logistic Regression, and Neural Networks). We use the StandardScaler fitted on the training data to transform the test data. This ensures that the test data is on the same scale as the training data, preventing the model from being biased by larger feature values. It's essential to use the same scaler to avoid introducing information from the test set into the training process, which would lead to an overestimation of the model's performance.

g. **Encoding**: If categorical variables in the training data were encoded, the same encoding scheme must be applied to the test data. This ensures that the model receives input in the expected format.
    i. **Explanation**: Machine learning models require numerical input. If the training data contains categorical features, they need to be converted into a numerical format (e.g., using one-hot encoding or label encoding). The same encoding method used for the training data must be applied to the test data to maintain consistency. For example, if we used one-hot

encoding for a "Color" feature in the training set, we need to apply the same one-hot encoding to the "Color" feature in the test set. This ensures that the model is expecting the same format of input features.

h. **Handling Missing Values**: If any missing values were imputed in the training data, the same imputation method should be applied to the test data.
    i. **Explanation**: Missing data is a common issue in real-world datasets. If we handled missing values in the training data (e.g., by replacing them with the mean, median, or a constant value), we need to use the same strategy for the test data. For instance, if we replaced missing "Age" values in the training set with the median age, we must replace the missing "Age" values in the test set with the *same* median age calculated from the training set. This ensures that the model is evaluated on data that has been processed in the same way as the training data.
i. **Feature Consistency**: The test dataset should contain the same features as the training dataset. Any features added or removed during feature engineering should be reflected in both datasets.
    i. **Explanation**: The model is trained to learn relationships between specific features and the target variable. The test data must contain the same set of features as the training data for the model to make accurate predictions. If we create new features or remove existing ones during feature engineering, these changes must be applied to both the training and test sets. This ensures that the model receives the expected input features during the prediction phase.

Model Application

Describe how the trained model was applied to the test dataset. Include code snippets if applicable.

j. **Explanation**: After preprocessing the test data, the trained machine learning model is used to predict the target variable for the test set. This process involves feeding the preprocessed test data into the model, which then outputs the predicted values. For classification problems, these predictions are typically class labels. The following code snippet demonstrates how to apply a trained model (in this case, an XGBoost model) to the test data:

```
# Assuming 'xgb_model' is the trained XGBoost model
y_pred_test = xgb_model.predict(X_test_scaled)
```

In this snippet:

i. xgb_model: This is the trained XGBoost model. It has learned the relationship between the input features and the target variable from the training data.
ii. X_test_scaled: This is the preprocessed test data, scaled using the same scaler that was applied to the training data. It contains the features for which we want to predict the target variable.
iii. y_pred_test: This variable stores the predicted target variable values for the test data. The predict() method of the trained model generates these predictions.

Test Metrics

Present the metrics used to evaluate the model's performance on the test dataset. Compare these results with the training and validation metrics.

k. **Explanation**: To evaluate the performance of the refined model on the test dataset, we use the same metrics that were used during the model refinement process. These metrics provide a comprehensive understanding of how well the model generalizes to unseen data. Here's a list of the key metrics and their interpretation:
    i. **Accuracy**: The proportion of correctly predicted instances out of the total number of instances.
        1. *Interpretation*: A higher accuracy indicates that the model makes more correct predictions overall. However, accuracy can be misleading if the classes are imbalanced.
    ii. **Precision**: The proportion of correctly predicted positive instances out of all instances predicted as positive.
        1. *Interpretation*: High precision indicates that when the model predicts a positive instance, it is likely to be correct.
    iii. **Recall (Sensitivity)**: The proportion of correctly predicted positive instances out of all actual positive instances.
        1. *Interpretation*: High recall indicates that the model is good at identifying most of the actual positive instances.
    iv. **F1-score**: The harmonic mean of precision and recall.
        1. *Interpretation*: The F1-score provides a balanced measure of precision and recall. It is useful when the classes are imbalanced.

v. **Confusion Matrix**: A table that summarizes the performance of a classification model by showing the counts of true positive, true negative, false positive, and false negative predictions.
    1. *Interpretation*: The confusion matrix provides detailed insights into the types of errors made by the model. It can help identify if the model is confusing certain classes with others.

By comparing the test set metrics with the training and validation metrics, we can assess how well the model generalizes to unseen data and identify any potential issues with overfitting or underfitting.

vi. If the test set performance is significantly lower than the training set performance, it suggests that the model may be overfitting to the training data.
vii. If the test set performance is similar to or slightly lower than the validation set performance, it indicates that the model is generalizing well.
viii. If the test set performance is low compared to both the training and validation sets, it suggests that the model may be underfitting.

Model Deployment

If applicable, discuss any steps taken to deploy the model in a real-world setting. This may include integration with other systems or platforms.

l. **Explanation**: Model deployment refers to the process of making a trained machine learning model available for use in real-world applications. This often involves integrating the model into an existing software system or creating a new system that utilizes the model. The specific steps involved in model deployment can vary significantly depending on the application, the infrastructure, and the requirements of the project. Here are some common deployment scenarios and considerations:
    i. **Web Application**: The model can be deployed as a web service, where it can be accessed via HTTP requests. This allows other applications to send data to the model and receive predictions in real-time. Frameworks like Flask (Python) and Django (Python) are commonly used for deploying models as web services.
    ii. **Mobile Application**: The model can be deployed directly on a mobile device, allowing the application to make predictions without requiring a network connection. This is often achieved using specialized libraries like TensorFlow Lite (for Android and iOS).

iii. **Embedded Systems**: For applications involving IoT devices, the model may need to be deployed on resource-constrained hardware. This requires careful optimization of the model to reduce its size and computational requirements.
iv. **Batch Processing**: In some cases, the model is used to make predictions on large volumes of data in an offline setting. This can be done using batch processing frameworks like Apache Spark.

Key considerations during model deployment include:

m. **Scalability**: The deployment infrastructure should be able to handle the expected volume of requests or data.
n. **Latency**: For real-time applications, it's crucial to minimize the time it takes for the model to generate predictions.
o. **Reliability**: The deployed model should be robust and fault-tolerant.
p. **Monitoring**: It's important to monitor the model's performance in production to detect any degradation in accuracy or other issues.
q. **Security**: The deployment environment should be secure to protect the model and the data it processes.

Code Implementation

Code Implementation

Include relevant code snippets for both model refinement and test submission phases. Use comments to explain key sections of the code.

Model Refinement

```
# Scaling numerical features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)

# Encoding the target variable
label_encoder = LabelEncoder()
y_train_encoded = label_encoder.fit_transform(y_train)
y_test_encoded = label_encoder.transform(y_test) # Ensure consistent encoding

# Model training and evaluation (XGBoost example)
xgb_model = XGBClassifier(use_label_encoder=False, eval_metric='mlogloss') #suppress a warning
```

```
xgb_model.fit(X_train_scaled, y_train_encoded) # Use encoded target
y_pred_xgb = xgb_model.predict(X_test_scaled)
accuracy_xgb = accuracy_score(y_test_encoded, y_pred_xgb)
print(f"XGBoost Accuracy: {accuracy_xgb:.2f}")
print("XGBoost Classification Report:")
print(classification_report(y_test_encoded, y_pred_xgb, target_names=label_encoder.classes_))
```

Test Submission

```
# No separate test submission phase in this notebook, the testing is incorporated in the evaluation during refinement.
# If there were a separate submission, you'd typically:
# 1. Retrain the best model on the *entire* training set.
# 2. Use that model to predict on the separate test set.
# 3. Format the predictions for submission.
#
# Example (conceptual):
# final_model = XGBClassifier(**best_xgb_params) # Use best params from tuning
# final_model.fit(np.concatenate((X_train_scaled, X_test_scaled)), # Combine data
# np.concatenate((y_train_encoded, y_test_encoded)))
# test_predictions = final_model.predict(X_submission_data_scaled) # Assuming you have X_submission_data
# formatted_submission = pd.DataFrame({'Vulnerability_Level': label_encoder.inverse_transform(test_predictions)})
# formatted_submission.to_csv('submission.csv', index=False)
```

Conclusion

The model refinement phase focused on optimizing the performance of an initial machine learning model by addressing issues like underfitting and overfitting. Techniques such as hyperparameter tuning, feature engineering, model selection, and cross-validation were employed. The XGBoost model was a key focus, with hyperparameter tuning leading to improved accuracy and reduced overfitting.

The test submission phase involved preparing the refined model for final evaluation. The best-performing model (XGBoost) was retrained on the entire training dataset, and the test dataset was prepared with the same preprocessing steps as the training data. The model's performance on

the test data was evaluated using metrics like accuracy, precision, recall, F1-score, and the confusion matrix.