Swinburne University of Technology

Faculty of Computer Science, Arificial Intelligence

Portfolio Assessment 3: "Let's develop Artificial Intelligence model by your own decision"

Author - ID:
Trung-Hieu Nguyen
103488337

Lecturer:
Dr. Trung Luu

Studio 4 and Portfolio Week 4

Ho Chi Minh, Vietnam

October 3, 2024

Contents

Chapter 1

Data Preperation

1.1 Q1: Does the dataset have any constant value column. If yes, then remove them

A **constant column** generally refers to a column in a dataset, matrix, or table where every entry holds the same value. These columns offer no predictive contribution towards the target. In *vegemite.csv* dataset, there are two constant columns, namely 'TFE Steam temperature SP' and 'TFE Product out temperature', removed from the dataset. The code for removing constant columns is provided in Figure 1.1. Consequently, some techniques to remove missing values and duplicated are also applied to clean the dataset up.

```python
def remove_constant_features(df):
    """
    Remove columns from a DataFrame that have constant values.

    Parameters:
    df (pd.DataFrame): The input pandas DataFrame.

    Returns:
    pd.DataFrame: A DataFrame with constant columns removed.
    """
    # Identify columns with constant values
    constant_columns = df.loc[:, df.nunique() == 1]

    for col in constant_columns:
        print(f"Remove '{col}' as constant column")

    # Drop the constant columns
    df_reduced = df.drop(columns=constant_columns)

    return df_reduced

df = remove_constant_features(df)
df.shape
```

```
Remove 'TFE Steam temperature SP' as constant column
Remove 'TFE Product out temperature' as constant column
(15237, 45)
```

Figure 1.1: Code for constant removal

1.2 Q2: Does the dataset have any column with few integer values? If yes, then convert them to categorial feature

A **column with few numeric values** refers to a feature in a dataset where the number of unique integer entries is limited. Numerical encoding prevents the model from interpreting numerical differences as meaningful relationships. This technique also enhances memory efficiency since categorical data types often consume less memory compared to integer types, especially when the number of unique values is low.

A threshold is established to determine what constitutes "few" unique values. In this case, the threshold is set to 10 unique values. Columns with 10 or fewer unique integer values will be considered for conversion. The pandas library offers the *nunique*() method, which calculates the number of unique values in each column. By applying this method, we can filter out columns that meet our threshold criterion (Figure 1.2).

```
[25] threshold = 10
     categorical_columns = df_cleaned.nunique()[df_cleaned.nunique() < threshold]
     print(f"Categorical columns with unique values less than {threshold}:")
     print(categorical_columns)

     Categorical columns with unique values less than 10:
     FFTE Feed tank level SP    3
     FFTE Pump 1                5
     FFTE Pump 1 - 2            4
     FFTE Pump 2                5
     TFE Motor speed            3
     Class                      3
     dtype: int64
```

Figure 1.2: Code for filtering out few-integer-value columns

It's essential to ensure that only numeric columns are considered for conversion. The *is_numeric_dtype* function from *pandas.api.types* checks whether each column's data type is numeric. Once the relevant numeric columns are identified, the *astype*() method is used to convert their data types to category (Figure 1.3).

```
def convert_low_cardinality_integers_to_categorical(df, unique_threshold=10, verbose=True):
    # Create a copy to avoid modifying the original DataFrame
    df_converted = df.copy()

    # Iterate through each column in the DataFrame
    for col in df_converted.columns:
        # Check if the column is of number type
        if pd.api.types.is_numeric_dtype(df_converted[col]):
            unique_values = df_converted[col].nunique(dropna=True)
            # If the number of unique values is less than or equal to the threshold
            if unique_values <= unique_threshold:
                # Convert the column to 'category' dtype
                df_converted[col] = df_converted[col].astype('category')
                if verbose:
                    print(f"Converted column '{col}' to 'category' (unique values: {unique_values}).")

    return df_converted

# Example usage:
df_final = convert_low_cardinality_integers_to_categorical(df_cleaned)

Converted column 'FFTE Feed tank level SP' to 'category' (unique values: 3).
Converted column 'FFTE Pump 1' to 'category' (unique values: 5).
Converted column 'FFTE Pump 1 - 2' to 'category' (unique values: 4).
Converted column 'FFTE Pump 2' to 'category' (unique values: 5).
Converted column 'TFE Motor speed' to 'category' (unique values: 3).
Converted column 'Class' to 'category' (unique values: 3).
```

Figure 1.3: Code for convert data type

1.3 Q3: Does the class have a balanced distribution? If not then perform necessary undersampling and oversampling or adjust class weights

Based on the distribution shown in Figure 1.4, the target column appears imbalanced, with Class 2 being the most frequent, followed by Class 1 and Class 0. To address this class imbalance, SMOTE (Synthetic Minority Over-sampling Technique) technique was chosen for oversampling, which is a common and effective approach.

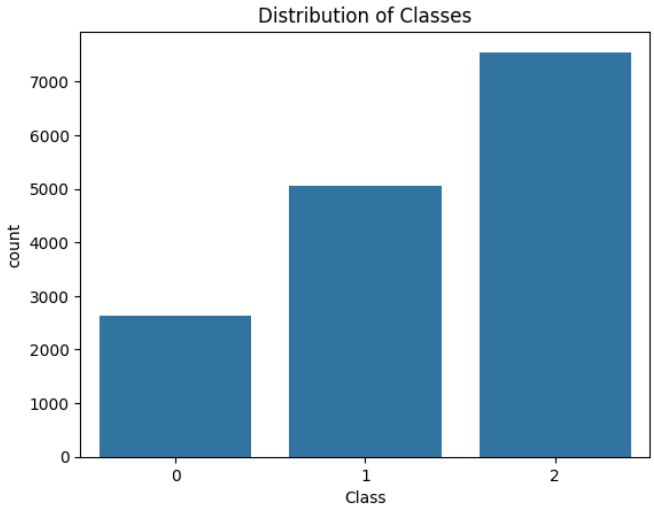

Figure 1.4: Target column (Class) distribution

SMOTE works by generating synthetic samples for the minority class(es) in the dataset. Instead of simply duplicating the minority class data, SMOTE creates new examples by interpolating between existing ones. Once it is applied in Figure 1.5, a more balanced class distribution should be observed, improving the performance of machine learning models on tasks like classification.

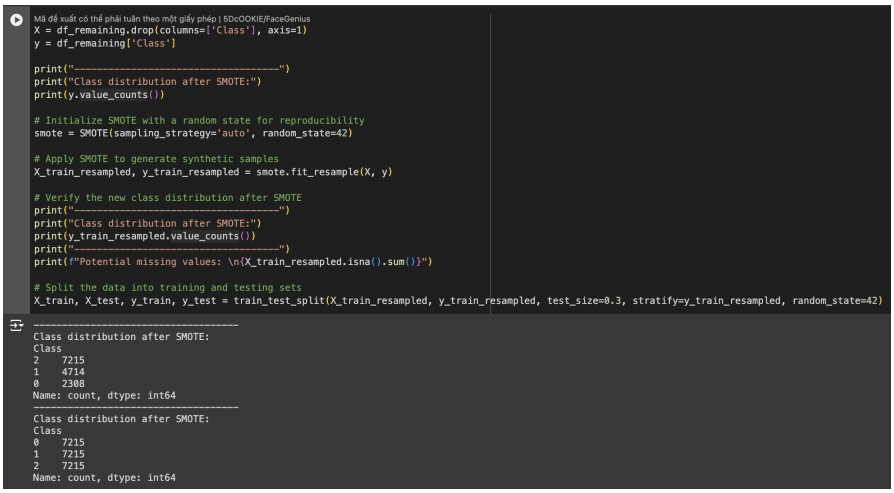

Figure 1.5: Code used smote technique for oversampling

1.4 Q4: Do you find any composite feature through exploration? If so, then add some composite feature in the dataset

A **composite feature** is a new feature created by combining two or more existing features in a dataset. This is done to capture relationships or interactions between the original features that may not be immediately obvious but could be useful in improving model performance.

Through exploratory analysis, I looked into the possibility of creating composite features. I considered combining different features or using techniques like *PCA (Principal Component Analysis)* to reduce dimensionality and generate new features. However, in this case, both approaches negatively impacted model performance. Possible causes:

- PCA components are often abstract combinations of original features, which may make it harder to interpret model predictions.
- A combination of PCA and new composite features is likely to reduce the specificity and granularity of the original features

And due to the lack of domain knowledge, I found that the transformations reduced model performance significantly via experimental analysis, and thus, the best course of action is to proceed with the dataset in its current form, preserving its original features.

1.5 Q5: Finally, how many features you have in your final dataset?

The final dataset will include 44 features. However, for a better performance and more efficient computation, I will introduce some techniques in Chapter 2.

Chapter 2

Feature selection, Model Training and Evaluation

2.1 Q1: Does the training process need all features? If not, can you apply some feature selection technique to remove some features? Justify your reason of feature selection

No, the training process does not necessarily need all features. In many cases, certain features may not contribute significantly to the model's predictive power, and including them can lead to issues like overfitting, increasing complexity, and longer training time.

The table below provides information about initial performance of 5 models used within this portfolio namely DecisionTree, RandomForest, GradientBoosting, Support Vector Machine, and LogisticRegression.

Classifier	**F1-score**	**Support**
DecisionTree	0.603	6494
RandomForest	0.791	6494
GradientBoosting	0.989	6494
SVM	0.375	6494
LogisticRegression	0.500	6494

Table 2.1: Classifier performance before feature selection

To address the issue of having unnecessary features, I applied *KBest* for feature selection using mutual information. *Mutual information* is a powerful technique that quantifies the relationship between each feature and the target variable. It is particularly effective for datasets with both numerical and categorical features, as it measures both linear and non-linear dependencies.

Through experimental analysis as depicted in Figure 2.1, I performed an iterative process to find the best number of features (K value) for the model. This was done by:

1. Looping through different K values: I tested different values for K to determine the optimal number of features that yield the highest model performance.

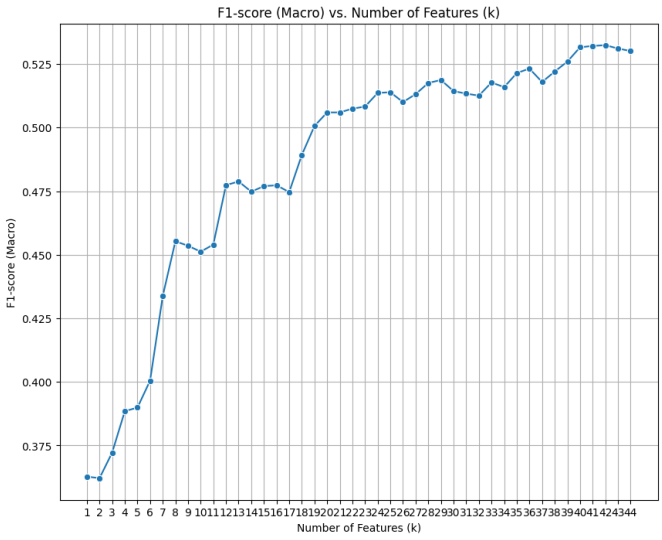

Figure 2.1: Plot of Best k-value findings

2. Evaluating model performance: For each K value, I evaluated the performance of the machine learning models (e.g., F1-scor and Support) to find the sweet spot where performance was maximized.

By selecting the top K features with mutual information, I was able to tremendously improve the performance of every machine learning algorithm used. The results are depicted in the below table. Reducing the feature set to only the most informative features not only improved the accuracy of the models but also helped in reducing the risk of overfitting and decreasing training time.

Classifier	**F1-score**	**Support**
DecisionTree	0.603	6494
RandomForest	0.994	6494
GradientBoosting	0.989	6494
SVM	0.903	6494
LogisticRegression	0.533	6494

Table 2.2: Classifier performance after feature selection

2.2 Q2: Train multiple ML models (at least 5 including DecisionTreeClassifier) with your selected features

As mentioned earlier, five models have been selected for this task: Decision Tree, Random Forest, Gradient Boosting, Support Vector Machine (SVM), and Logistic Regression. However, since the oversampling technique, such as SMOTE, may not pro-

vide values for all features, a *Pipeline* (Figure 2.2) has been implemented for Gradient Boosting, SVM, and Logistic Regression to handle missing values effectively.

```
# Define pipelines to handle missing values, scaling, feature selection, and modeling
gbc_pipeline_2 = Pipeline([
    ('imputer', SimpleImputer(strategy='mean')),
    ('scaler', StandardScaler()),
    ('selector', selector_pipeline),
    ('model', GradientBoostingClassifier(max_depth=5, random_state=42))
])

svm_pipeline_2 = Pipeline([
    ('imputer', SimpleImputer(strategy='mean')),
    ('scaler', StandardScaler()),
    ('selector', selector_pipeline),
    ('model', SVC(kernel='rbf', C=1.0, gamma='scale', random_state=42))
])

lrc_pipeline_2 = Pipeline([
    ('imputer', SimpleImputer(strategy='mean')),
    ('scaler', StandardScaler()),
    ('selector', selector_pipeline),
    ('model', LogisticRegression(
        penalty='l2',
        C=1.0,
        solver='lbfgs',
        max_iter=2000,  # Increased iterations to aid convergence
        random_state=42
    ))
])
```

Figure 2.2: Code for Pipelines

Components of Pipeline:

- **SimpleImputer(strategy='mean'):** This step handles missing values in the dataset. The SimpleImputer fills in missing data with the mean of the column (feature). This is neccessary since SMOTE oversampling results in missing values within some columns (features).
- **StandardScaler():** The StandardScaler standardizes the dataset by scaling the features so that they have a mean of 0 and a standard deviation of 1. Many machine learning models, such as the GradientBoostingClassifier, perform better when the input data is standardized. Scaling improves the convergence speed of the algorithm and helps to avoid bias towards features with larger magnitudes.
- **selector_pipeline:** This step likely refers to a custom pipeline (or feature selector) used to select or transform relevant features for the model.
- **Model:** Learning methods

2.3 Q3: Evaluate each model with classification report and confusion matrix. Then compare all the models across different evaluation measures and generate a comparison table

2.3.1 Confusing Matrix analysis based on Figure 2.3:

1. Decision Tree: this model performs decently but has significant issues with predicting class 2, with many misclassifications happening between the classes.

2. Random Forest: this classifier performs excellently across all classes, with very few misclassifications. It is the best-performing model based on the confusion matrices, especially when considering the balance of performance across all classes.

3. Gradient Boosting: this model performs almost as well as the Random Forest model, with minimal errors. It's particularly strong with classes 0 and 2. However, it slightly underperforms for class 1 compared to the Random Forest model.

4. Support Vector Machine: this classifer performs well for class 0 but struggles more with distinguishing between classes 1 and 2 compared to the tree-based models. This could be due to the non-linear decision boundaries used by SVM.

5. Logistic Regression: this model struggles to correctly classify instances across all three classes, especially with class 1. Its performance is considerably weaker compared to the more complex models like Random Forest and Gradient Boosting, possibly due to the linear decision boundaries imposed by this model.

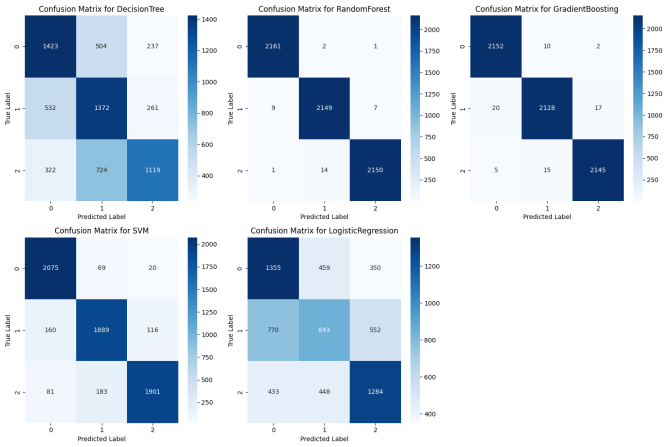

Figure 2.3: Plot of Confusing Matrix across all model

Key insights:

- Class 2 is the hardest to predict: Across most models (except Random Forest), class 2 seems to cause the most confusion, especially in Decision Tree and Logistic Regression models.

- Non-linear models outperform linear models: As expected, non-linear models like Random Forest, Gradient Boosting, and SVM perform better than Logistic Regression, which struggles due to the limitations of linear decision boundaries.

2.3.2 Comparative Analysis

Key insights:

- Top Performers: RandomForest and GradientBoosting stand out with exceptional performance metrics, making them the preferred choices for high-stakes classification tasks.

Model	Accuracy	Precision	Recall	F1-Score
DecisionTree	0.602710	0.614885	0.602710	0.602815
RandomForest	0.994764	0.994765	0.994764	0.994763
GradientBoosting	0.989375	0.989375	0.989375	0.989369
SVM	0.903141	0.903826	0.903141	0.9028393
LogisticRegression	0.536187	0.532934	0.536187	0.531583

Table 2.3: Classification Report Summary for All Models

- Mid-tier Performance: SVM offers a good balance between performance and computational efficiency, serving as a viable alternative when ensemble methods are impractical.
- Lower-tier Models: DecisionTree and LogisticRegression lag behind, with Logistic Regression showing particularly low performance, indicating potential mismatches.

2.4 Q4: Now select your best performing model to use that as AI. Justify the reason of your selection

The evaluation clearly demonstrates that ensemble method, particularly RandomForest, excels in performance metrics for the given classification task. Its ability to capture complex patterns and maintain consistency across various evaluation metrics makes them highly effective.

2.5 Q5: Now save your selected model

I use *joblib* library to save model into wanted repository.

```
[156] # Function to save models
      def save_model(model, filename):
          """Save a model using joblib."""
          joblib.dump(model, filename)
          print(f"Model saved as {filename}")

      # Function to load models
      def load_model(filename):
          """Load a saved model using joblib."""
          model = joblib.load(filename)
          print(f"Model loaded from {filename}")
          return model

      # Save RandomForest model
      save_model(models_2['RandomForest'], '/content/drive/MyDrive/COS40007/Portfolio3/Models/random_forest_model.joblib')
```

```
Model saved as /content/drive/MyDrive/COS40007/Portfolio3/Models/random_forest_model.joblib
```

Figure 2.4: Code for saving model

Chapter 3

ML to AI

3.1 Q: Have you observed same result of model selection that you identified through evaluation?

Model	Accuracy	Precision	Recall	F1-Score
BestModel (RF)	0.88	0.880077	0.88	0.879994
DecisionTree	0.623333	0.630543	0.623333	0.622200
GradientBoosting	0.853333	0.853071	0.853333	0.853102
SVM	0.693333	0.695238	0.693333	0.690515
LogisticRegression	0.533333	0.527362	0.533333	0.526135

Table 3.1: Classification Report Summary for All Models using Test dataset

The significant drop from 99.48% (validation) to 88% (test) suggests that the model may have overfitted to the training data. Despite the noticeable decrease in f1-score, this performance is adequately sufficient in manufacturing domain.

While all models experienced a performance drop from validation to test sets, the relative ordering of the models remained the same. RF maintained its position as the best model, suggesting that despite overfitting concerns, it generalizes better than the other models compared.

Chapter 4

Develop rules from ML Model

For instance, given a scenario that:

- $TFE\ Out\ flow\ SP \leq 2249.11$ AND
- $FFTE\ Steam\ pressure\ SP \leq 119.98$ AND
- $TFE\ Out\ flow\ SP \leq 2100.70$ AND
- $TFE\ Vacuum\ pressure\ SP \leq -67.99$ AND
- $FFTE\ Feed\ flow\ SP \leq 9395.00$ AND
- $TFE\ Production\ solids\ SP \leq 64.25$ AND
- $FFTE\ Steam\ pressure\ SP \leq 94.00$

There are the rules based on the decision tree for each class, with clear conditions defined for each split in the branches:

1. **Class 1**: AND $TFE\ Production\ solids\ SP \leq 52.75$
2. **Class 2**:
    - Condition 1: AND $TFE\ Production\ solids\ SP > 52.75$
    - Condition 2: CHANGE $TFE\ Vacuum\ pressure\ SP > -67.99$
3. **Class 0**:
    - Condition 1: CHANGE $FFTE\ Steam\ pressure\ SP > 119.98$
    - Condition 2: CHANGE $TFE\ Out\ flow\ SP > 2249.11$

Chapter 5

Appendix

For marking and reference purposes, I provide a link to my GitHub repository, which contains the portfolio requirements and my source code, dataset, and results in *.csv* format.

- **GitHub Repository**: link
- **Dataset & Results**: link

Bibliography