Project 1 Report

Explain the concept and code detail of my custom preprocessor class in pre processor.py (40%):

Concept and code explanation for data preprocessing

The following private methods are used by the public method `preprocessing` , my goal in this part is to preprocess the data given as a data frame so that the prediction model can have a better performance.

1. `_drop_index`

```
def _drop_index(self):
    if 'Unnamed: 0' in self.df.columns:
        self.df = self.df.drop(['Unnamed: 0'], axis=1)
    return self
```

This method is designed to drop the index column in a data frame. Since it is required that train and test data should not contain index, this method checks all the columns of a given data frame and drops the column with feature named 'Unnamed: 0', i.e., the index column.

2. `standardize_scalar`

```
def _standardize_scalar(self):
    for col in self.df.columns:
        if self.df[col].dtype == 'object':
            self.df[col] = pd.Categorical(self.df[col]).codes
    return self
```

This method is used to transform data from columns F18 to F77. They are converted from 'Yes', 'No', NaN to 1, 0, -1 respectively. This step is necessary since the model can only handle numerical values.

3. `_mice_imputation`

```python
def _mice_imputation(self, iterations=10):
    for column in self.df.columns:
        if self.df[column].isnull().sum() > 0:
            missing_indices = self.df[self.df[column].isnull()].index
            for idx in missing_indices:
                self.df.loc[idx, column] = self.df[column].mean()
    for _ in range(iterations):
        for column in self.df.columns:
            if self.df[column].isnull().sum() > 0:
                correlations = self.df.corr()[column].abs().sort_values(ascending=False)
                top_correlated = correlations[1:11].index.tolist()
                missing_indices = self.df[self.df[column].isnull()].index
                for idx in missing_indices:
                    row_values = self.df.loc[idx, top_correlated].dropna()
                    weights = correlations[row_values.index]
                    imputed_value = (row_values * weights).sum() / weights.sum()
                    self.df.loc[idx, column] = imputed_value
    return self
```

This method implements MICE imputation to handle missing values. It runs for a specific number of iterations to improve the imputation quality. Instead of using all columns, the method selects the top 10 most correlated features to calculate the imputed values. Here's the process:

- Initialize all the missing value as the mean of each column.
- The method runs for a defined number of iterations.
- In each iteration, it checks for missing values in the columns. If a column has missing values, it computes the correlation between that column and all other columns, then selects the top 10 most correlated features.
- For each missing value, the method looks at the values in the top correlated columns from the same row.
- Finally, the missing value is imputed using the weighted average of the top correlated features.

4. `_remove_outliers`

```python
def _remove_outliers(self, columns, threshold=3):
    for col in columns:
        if self.df[col].dtype in ['int64', 'float64']:
            z_scores = np.abs((self.df[col] - self.df[col].mean()) / self.df[col].std())
            self.df = self.df[z_scores < threshold]
    return self
```

This method removes the outliers by the Z-score method, so that the training data will not be effected by the skewed data.

- What are the steps of the Z-score method.
    1. The method computes the Z-scores for each columns.
    2. Filter out rows where the Z-scores exceed the threshold (default is 3).

5. `_add_polynomial_features`

```python
def _add_polynomial_features(self, columns, degree=2):
    for col in columns:
        for d in range(2, degree + 1):
            new_col_name = f"{col}^2"
            self.df[new_col_name] = self.df[col] ** d
    return self
```

This method loops through each column, perform power operations and stores the results in as new columns.

- Why do we need to add polynomial features?

    Adding polynomial features in a training data set can improve the model to capture the non-linear relationships in the dataset.

6. `_normalize`

```python
def _normalize(self, numeric_columns):
    self.df[numeric_columns] = (
        (self.df[numeric_columns] - self.df[numeric_columns].mean())
        / self.df[numeric_columns].std()
    )
    return self
```

This method normalizes all the numeric columns of the data frame by the formula below:

$$Z = \frac{x - \mu}{\sigma}$$

- Why do we need to normalize the data?
    - to ensure that all features contribute equally to model training
    - to make sure that the results of the training are not varied by the unit of the each columns

The public method `Preprocess`

```python
def preprocess(self):
    self._drop_index()
    self._standardize_scalar()
    self._mice_imputation()
    numeric_columns = self.df.select_dtypes(include=['int64', 'float64']).columns
    self._remove_outliers(numeric_columns)
    self._add_polynomial_features(numeric_columns)
    updated_numeric_columns = self.df.select_dtypes(include=['int64', 'float64']).columns
    self._normalize(updated_numeric_columns)

    y = (self.df['y'] > 0.5).astype(int).to_numpy()
    X = self.df.drop('y', axis=1).to_numpy(dtype=float)
    return X, y
```

Finally, I combined all the private methods explained above into this public method. It operates in the following order:

1. Dropping irrelevant columns (`_drop_index`).
2. Standardizing scalars (`_standardize_scalar`).
3. Filling missing values with MICE (`_mice_imputation`).
4. Removing outliers (`_remove_outliers`).
5. Adding polynomial features (`_add_polynomial_features`).
6. Normalizing numeric features (`_normalize`).

In the final step, we ensure that:

- y is binary
- y is separated from X
- both X and y are returned as `numpy` arrays

Explain the concept and code detail of your dataPreprocessing function in main.py (10%):

```python
def dataPreprocessing():
    train_data = pd.read_csv('train_X.csv')
    train_label = pd.read_csv('train_y.csv')
    train_data['y'] = train_label.iloc[:, -1]

    train_X, train_y = Preprocessor(train_data).preprocess()

    test_data = pd.read_csv('test_X.csv')
    test_labels = pd.read_csv('test_y.csv')
    test_data['y'] = test_labels.iloc[:, -1]

    test_X, test_y = Preprocessor(test_data).preprocess()

    return train_X, train_y, test_X, test_y
```

The `dataPreprocessing()` function loads, processes both training and testing datasets in the same way. Note that the training and testing datasets are preprocessed independently, so the data leaking issue is not possible to happen. The following are the steps of processing the datasets:

1. Load training data

```
train_data = pd.read_csv('train_X.csv')
train_label = pd.read_csv('train_y.csv')
```

2. Combining features and labels

```
train_data['y'] = train_label.iloc[:, -1]
```

We combined the label data into the `train_data` DataFrame. This step makes it easier to handle data transformations that affect both features and labels simultaneously.

3. Preprocessing the testing data

```
train_X, train_y = Preprocessor(train_data).preprocess()
```

Here, we are using the `Preprocessor` class defined in the first section to preprocess the training datasets train_X and train_y.

4. Load testing data

```
test_data = pd.read_csv('test_X.csv')
test_label = pd.read_csv('test_y.csv')
```

5. Combining features and labels

```
test_data['y'] = test_label.iloc[:, -1]
```

Same as step 2, we combine the testing data with the testing label.

6. Preprocessing the testing data

```
train_X, train_y = Preprocessor(train_data).preprocess()
```

Same, as step 3, we are using the `Preprocessor` class defined in the first section to preprocess the testing datasets test_X and test_y.

7. Returning preprocessed data

```
    return train_X, train_y, test_X, test_y
```

After the preprocess is done, the function returns the preprocessed feature matrices (`train_X` and `test_X`) and corresponding target labels (`train_y` and `test_y`) for both training and test sets.

Why is the same preprocessing method applied to both training and testing data?

1. Ensure both datasets are on the same scale and format
2. Improving model performance: the model can better adapt to the types of data it will encounter during deployment

Discussion (10%):

Feature Selection

While working on the preprocessing part, I tried adding feature selection to the Preprocessor class.

- First, I tried to keep only the most important features. However, after testing it out, I found that this didn't improve the accuracy and actually lowered it.
- In my opinion, this happened because not all features are equally helpful, and reducing the number of features can sometimes remove important information that the model needs.

MICE Imputation

I used the MICE imputation method for filling in missing values.

- I discovered that having more top correlated features isn't always better. In my case, using top 10 correlated features generates the highest accuracy.
- It is important to choose the right number of features since we can avoid adding too much noise or irrelevant information.

Bonus (10%):

The implementation is explained above. Here, I would like to share an overview of MICE.

MICE is a method used to fill in missing values in a dataset. Instead of just guessing a single value for each missing spot, MICE improves its guess with every imputation in each iteration. First, MICE gives each missing value an initial guess. Then, it computes the weights of the columns based on rows with complete data. The imputed value is typically a weighted average of the related features, and the process continues until the guesses stabilize and don't change much between iterations.

Acc Rate Screenshot (25%)