Data Preprocessing for Classification Model

This document outlines the steps involved in preprocessing the UCI Adult Census dataset for a classification model. Each step is described with its rationale and relevant code examples.

1. Data Loading

```python
adult_df = pd.read_csv("adult.csv", names=["age", "workclass", "fnlwgt", "education", "education-num", "marital-status", "occupation", "relationship", "race", "sex", "capital-gain", "capital-loss", "hours-per-week", "native-country", "income"], skipinitialspace=True)
```

Column names are explicitly specified to ensure clarity and consistency in data manipulation. The `skipinitialspace=True` parameter is used to remove any leading whitespace from column names, thereby avoiding potential data parsing issues.

2. Splitting the Dataset into Features and Target

```python
X = adult_df.drop('income', axis=1)
y = adult_df['income']
```

The dataset is divided into feature variables (`X`) and the target variable (`y`). The `drop` method is employed to exclude the `'income'` column from `X`, which contains the feature data, while `y` contains the target variable indicating income classification.

3. Encoding Categorical Variables

```python
encoder = OrdinalEncoder()
X_encoded = encoder.fit_transform(X)
```

Categorical variables are encoded into numerical values using `OrdinalEncoder`. This method converts categorical features into ordinal numerical values based on their inherent order. Given that RandomForrest only works with numerical values, It is key to convert categorical values to numbers for processing.

4. Calculating Feature Importances

```python
rf = RandomForestClassifier(n_estimators=1000, random_state=42)
```

rf.fit(X_encoded, y)
importances = rf.feature_importances_
```

Feature importance scores are calculated using a `RandomForestClassifier`. This model is trained on the encoded features, and the `feature_importances_` attribute is used to retrieve the importance scores of each feature. Random forests are chosen for their robustness and ability to handle various types of data effectively. They are also not prone to overfitting and provide consistent results
Leaving “Gender” all other variable have a good correlation with the output and hence “Gender” is not considered in the final Training and prediction

## 5. Calculating Cumulative Importances

```**python**
cumulative_importances = np.cumsum(importances)
```

The cumulative sum of feature importances is computed to assess the total contribution of features incrementally. This information is useful for feature selection, as it helps determine how many features are necessary to reach a desired level of total importance.

## 6. Identifying the Number of Features for 95% Cumulative Importance

```**python**
num_features = np.where(cumulative_importances > 0.95)[0][0] + 1
```

The number of features required to account for at least 95% of the cumulative importance is determined. This threshold helps in reducing the dimensionality of the dataset by selecting a subset of features that capture the majority of the relevant information, thus enhancing model performance and interpretability.

## 7. Extracting the Most Important Features

```**python**
feature_names = list(X.columns)
important_feature_names = [feature_names[i] for i in np.argsort(importances)[-num_features:]]
important_features = X_encoded[:, np.argsort(importances)[-num_features:]]
```

The most important features are extracted based on their importance scores. The names of these features are obtained and stored in `important_feature_names`, while `important_features` contains the corresponding subset of the dataset. This step ensures that the model focuses on the most relevant features, which can improve both performance and interpretability.
```

8. Handling Data Skewness

```python
positive_class = np.where(y == '>50K')[0]
negative_class = np.where(y == '<=50K')[0]
```

The dataset was tilted in the direction of the “<=50K”, nearly 76% of all total values belong to this class and as such the basic training alone would give terrible results. As a remedy I over sampled the “>50K” to get things on a little more even ground. The operations were effective as the accuracy of the model moved up greatly. Specifics will be discussed in the Model Evaluation Document

9. Oversampling the Minority Class

```python
negative_class = np.random.choice(negative_class, size=len(positive_class), replace=True)
```

To handle the class imbalance, the minority class (`'>50K'`) is oversampled by randomly selecting instances from the negative class (`'<=50K'`) with replacement. This ensures that the number of instances in both classes is equal, thereby balancing the dataset. Oversampling helps in mitigating the bias towards the majority class and improves the classifier's ability to correctly predict the minority class.

10. Combining the Positive and Negative Classes

```python
X_train = np.concatenate([important_features[positive_class], important_features[negative_class]], axis=0)
y_train = np.concatenate([y[positive_class], y[negative_class]], axis=0)
```

The oversampled negative class instances are combined with the positive class instances to create a balanced training set. This is achieved by concatenating the feature matrices (`important_features`) and the target vectors (`y`). The resulting `X_train` and `y_train` datasets are now balanced, ensuring that the classifier receives equal representation from both classes during training. This step is crucial for building a robust model that performs well across both classes.