Machine Learning Analysis on AVC and Salary Datasets

Dragomir Andrei

May 25, 2024

Abstract

This document provides a comprehensive analysis of two machine learning models applied to the given datasets. The analysis includes data preprocessing, model training, evaluation, and comparison of results.

Contents

1 Introduction

In this analysis, we explore the Logistic Regression and the Multy Layer Perceptron models on the AVC and Salary datasets. The goal is to predict the target variable based on the features provided in the datasets.

Dataset Description

2 Medical and Lifestyle Information Dataset

This dataset contains medical values and relevant lifestyle information for 5110 individuals. The dataset is intended to be used for predicting whether a person is likely to have a cerebrovascular accident (stroke) or not. The target attribute is *cerebrovascular_accident*, which has binary values indicating the presence (1) or absence (0) of a stroke. The classification task is binary.

The attributes in the dataset are as follows:

Table 1: Attributes of the Medical and Lifestyle Information Dataset

Attribute	**Type**	**Description**
mean_blood_sugar_level	numeric	The average blood sugar level over the observation period.
cardiovascular_issues	categorical	Indicates if the subject has a history of cardiovascular issues. Possible values: 0, 1.
job_category	categorical	The domain in which the person works. Possible values: child, entrepreneurial, N_work_history, private_sector, public_sector.
body_mass_indicator	numeric	The body mass index indicating if the person is underweight, normal weight, overweight, or obese.
sex	categorical	The gender of the person. Possible values: F, M.
tobacco_usage	categorical	Indicator for tobacco use, present or past. Possible values: ex-smoker, smoker, non-smoker.
high_blood_pressure	categorical	Binary attribute indicating if a person suffers from high blood pressure. Possible values: 0, 1.
married	categorical	Binary attribute indicating if the person has ever been married. Possible values: Y, N.
living_area	categorical	Type of area where the person has lived most of their life. Possible values: City, Countryside.
years_old	numeric	The age of the person in years.
chaotic_sleep	categorical	Binary attribute for an irregular sleep schedule. Possible values: 0, 1.
analysis_results	numeric	Results of the person's medical tests, which may include various measurements and health indicators.
biological_age_index	numeric	An index estimating the biological age of a person based on various factors such as lifestyle and health status.
cerebrovascular_accident	categorical	Binary indicator of whether the person has had a stroke. Possible values: 0, 1.

3 Employee Information Dataset

This dataset contains personal, educational, and professional information of various employees. The objective of this dataset is the binary classification of employees into categories of earning above or below $50K per year. The classification task is binary.

The attributes in the dataset are as follows:

Table 2: Attributes of the Employee Information Dataset

Attribute	**Type**	**Description**
fnl	numeric	Socio-economic characteristic of the population from which the individual comes.
hpw	numeric	Number of hours worked per week.
relation	categorical	The type of relationship in which the individual is involved.
gain	numeric	Capital gain.
country	categorical	Country of origin.
job	categorical	The individual's occupation.
edu_int	numeric	Number of years of education.
years	numeric	Age of the individual.
loss	numeric	Capital loss.
work_type	categorical	Type of occupation.
partner	categorical	Type of partner the individual has.
edu	categorical	Type of education of the individual.
gender	categorical	Gender of the individual.
race	categorical	Race of the individual.
prod	numeric	Capital production.
gtype	categorical	Type of work contract.

4 Attribute Type Identification

Before utilizing a machine learning model for a dataset, it is crucial to identify the types of features in the dataset and their values. Understanding the nature of the attributes helps in selecting appropriate preprocessing techniques and models. The key distinctions among the types of attributes in the provided datasets are as follows:

4.1 Continuous Numerical Attributes

Continuous numerical attributes are features that can take any value within a given range. These attributes are measured on a continuous scale and can be divided into finer increments. Examples from the datasets include:

- **mean_blood_sugar_level**: Average blood sugar level measured over the observation period.
- **body_mass_indicator**: Body mass index indicating whether a person is underweight, normal weight, overweight, or obese.
- **years_old**: Age of the person in years.

- **analysis_results**: Results of medical tests.
- **biological_age_index**: An index estimating the biological age of a person based on various factors.
- **fnl**: Socio-economic characteristic of the population the individual comes from.
- **hpw**: Number of hours worked per week.
- **gain**: Capital gain.
- **edu_int**: Number of years of education.
- **years**: Age of the individual.
- **loss**: Capital loss.
- **prod**: Capital production.

4.2 Discrete Attributes

Discrete attributes are features that take on a finite number of distinct values. These values are often categorical and can be counted in whole numbers. Examples from the datasets include:

- **cardiovascular_issues**: Whether the subject has a history of cardiovascular issues (0 or 1).
- **sex**: Gender of the person (F or M).
- **tobacco_usage**: Indicator for smokers, either past or present (ex-smoker, smoker, non-smoker).
- **high_blood_pressure**: Indicator if a person has high blood pressure (0 or 1).
- **married**: Whether the person has ever been married (Y or N).
- **living_area**: Type of area where the person has lived most of their life (City or Countryside).
- **chaotic_sleep**: Indicator for an irregular sleep schedule (0 or 1).
- **cerebrovascular_accident**: Indicator if the person has had a stroke (0 or 1).
- **relation**: Type of relationship the individual is involved in.
- **country**: Country of origin.
- **job**: Job of the individual.
- **work_type**: Type of job.
- **partner**: Type of partner the individual has.
- **edu**: Type of education of the individual.

- **gender**: Gender of the individual.
- **race**: Race of the individual.
- **gtype**: Type of work contract.
- **job_category**: Domain in which the person works.

Recognizing these distinctions helps in choosing the right methods for handling the data during preprocessing. For instance, continuous numerical attributes might require normalization or standardization, discrete attributes might need encoding, and ordinal attributes might need ordinal encoding to maintain the order information. Properly identifying and classifying the attributes ensures that the machine learning models can effectively learn from the data and make accurate predictions.

5 Numeric Attributes Analysis

In this section, we analyze the numeric attributes of the two datasets provided. The analysis includes the number of non-missing values, mean value, standard deviation, minimum value, 25th percentile, 50th percentile (median), 75th percentile, and maximum value for each numeric attribute.

5.1 Healthcare Dataset

Table 3: Statistics of Numeric Attributes in the Healthcare Dataset

Attribute	**No-miss**	**Mean**	**Std Dev**	**Min**	**25th Pctl**	**Mid**	**75th Pctl**	**Max**
mean_blood_sugar_level	5110	106.15	45.28	55.12	77.25	91.89	114.09	271.74
body_mass_indicator	4909	28.89	7.85	10.30	23.50	28.10	33.10	97.60
years_old	5110	46.57	26.59	0.08	26.00	47.00	63.75	134.00
analysis_results	4599	323.52	101.58	104.83	254.65	301.03	362.82	756.81
biological_age_index	5110	134.78	50.40	-15.11	96.71	136.37	172.51	266.99

Comments:

- The **mean_blood_sugar_level** has a mean of 106.15 with a standard deviation of 45.28, indicating a wide range of values.
- The **body_mass_indicator** shows a mean of 28.89, suggesting that on average, individuals fall into the overweight category.
- The **years_old** attribute ranges from 0.08 to 134 years, with a median of 47 years.
- The **analysis_results** attribute has significant variability, as indicated by its standard deviation of 101.58.
- The **biological_age_index** has negative values, which may need to be investigated further for data correctness.

5.2 Employee Dataset

Table 4: Statistics of Numeric Attributes in the Employee Dataset

Attribute	**No-miss**	**Mean**	**Std Dev**	**Min**	**25th Pctl**	**Mid**	**75th Pctl**	**Max**
fnl	9999	190352.9	106070.8	19214	118282.5	178472	237311	1455435
hpw	9199	40.42	12.52	1.0	40.0	40.0	45.0	99.0
gain	9999	979.85	7003.80	0.0	0.0	0.0	0.0	99999.0
edu_int	9999	14.26	24.77	1.0	9.0	10.0	13.0	206.0
years	9999	38.65	13.75	17.0	28.0	37.0	48.0	90.0
loss	9999	84.11	394.04	0.0	0.0	0.0	0.0	3770.0
prod	9999	2014.93	14007.60	-28.0	42.0	57.0	77.0	200125.0

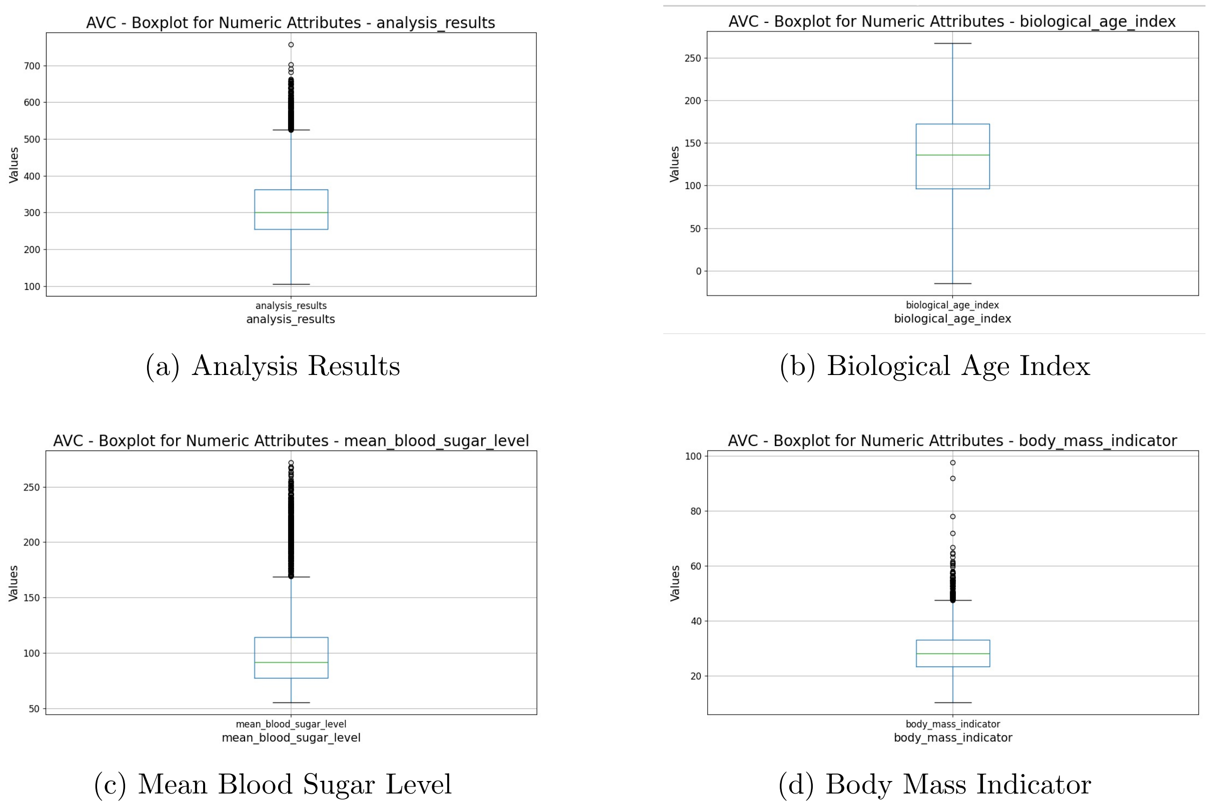
Comments:

- The **fnl** attribute has a high mean value and a wide range, indicating significant differences in the socio-economic status of individuals.
- The **hpw** (hours per week) attribute shows that most individuals work around 40 hours per week.
- The **gain** and **loss** attributes have a high standard deviation, indicating that only a few individuals have large capital gains or losses.
- The **edu_int** (years of education) attribute shows that the majority of individuals have around 10 to 13 years of education.
- The **years** attribute shows an average age of around 38.65 years, with a minimum of 17 and a maximum of 90 years.
- The **prod** attribute has a high standard deviation, indicating variability in capital production.

6 Numeric Ranges for Attributes

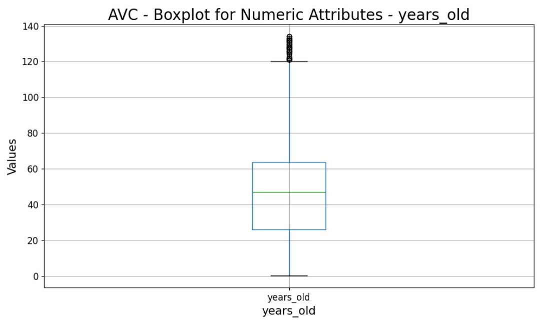
In this section, we present the boxplots for the numeric attributes in the datasets to visualize the distribution and identify any potential outliers.

6.1 Healthcare Dataset



(a) Analysis Results (b) Biological Age Index

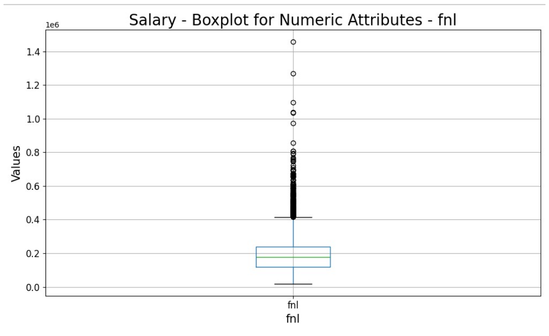
(c) Mean Blood Sugar Level (d) Body Mass Indicator



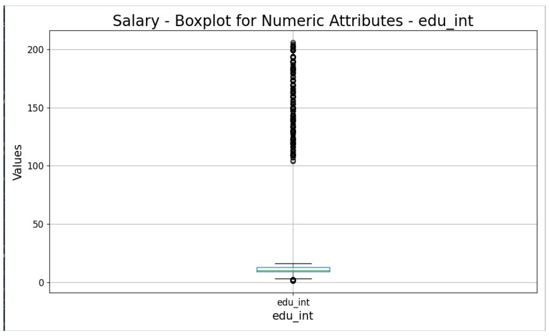
(e) Years Old

Figure 1: Boxplots for Healthcare Dataset Numeric Attributes

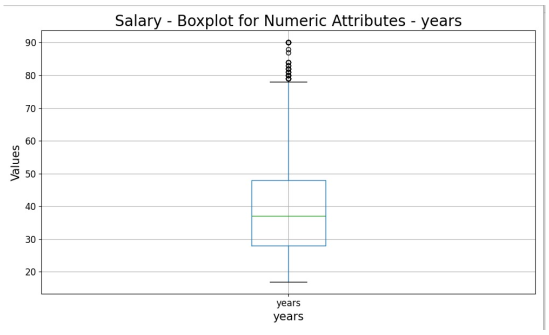
6.2 Employee Dataset



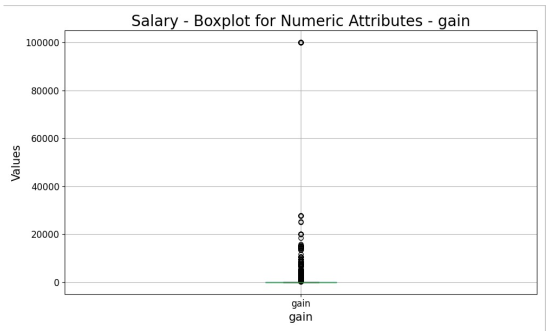
(a) FNL



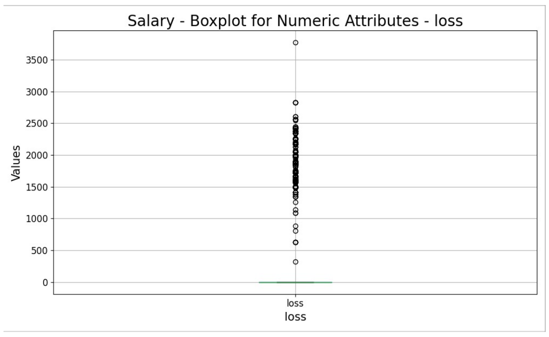
(b) Education (Years)



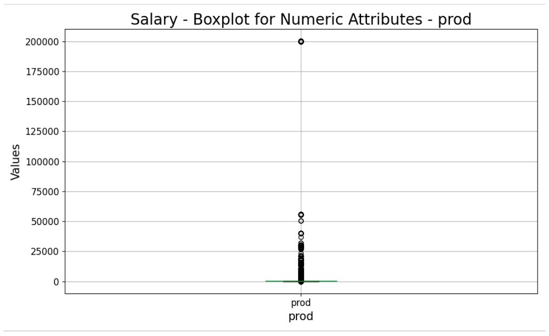
(c) Years



(d) Gain



(e) Loss



(f) Prod

Figure 2: Boxplots for Employee Dataset Numeric Attributes

7 Categorical Attributes Analysis

In this section, we analyze the categorical attributes of the datasets. The tables below present the attributes along with the number of non-missing values and the number of unique values for each attribute.

7.1 Healthcare Dataset

Table 5: Categorical Attributes - Healthcare Dataset

Attribute	**Number of Non-Missing Values**	**Number of Unique Values**
cardiovascular_issues	5110	2
job_category	5110	5
sex	5110	2
tobacco_usage	5110	4
high_blood_pressure	5110	2
married	4599	2
living_area	5110	2
chaotic_sleep	5110	2
cerebrovascular_accident	5110	2

7.2 Employee Dataset

Table 6: Categorical Attributes - Employee Dataset

Attribute	**Number of Non-Missing Values**	**Number of Unique Values**
relation	9999	6
country	9999	41
job	9999	14
work_type	9999	9
partner	9999	7
edu	9999	16
gender	9199	2
race	9999	5
gtype	9999	2
money	9999	2

7.3 Comments

The healthcare dataset includes several categorical attributes such as ‘cardiovascular_issues‘, ‘job_category‘, ‘sex‘, and ‘tobacco_usage‘. Each of these attributes has a varying number of unique values, indicating different levels of categorization. For instance, ‘tobacco_usage‘ has four unique values, while ‘sex‘ has only two. It is also noted that the ‘married‘ attribute has fewer non-missing values compared to others, which might require special handling during data preprocessing.

The employee dataset, on the other hand, contains a wider range of categorical attributes with a larger variety of unique values, especially for ‘country‘ and ‘edu‘, which have 41 and 16 unique values, respectively. Attributes like ‘gender‘ and ‘gtype‘ are binary, which simplifies their processing. The variety in the number of unique values across different attributes in both datasets indicates the complexity and diversity of the data, which must be carefully considered during the feature engineering and model building stages.

8 Categorical Distribution for Attributes

8.1 Healthcare Dataset

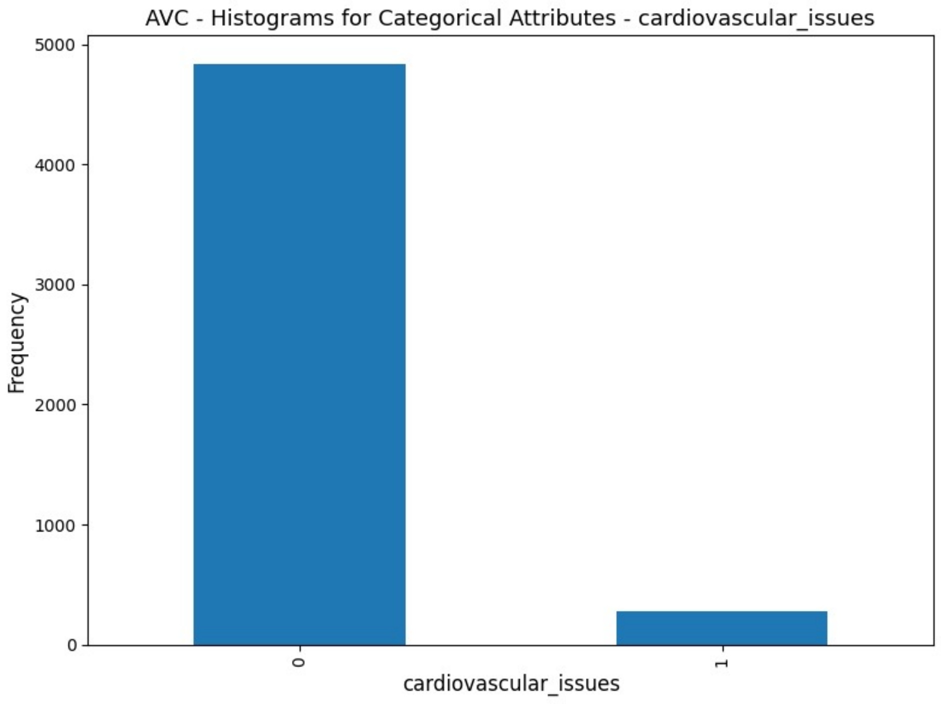


Figure 3: Cardiovascular Issues

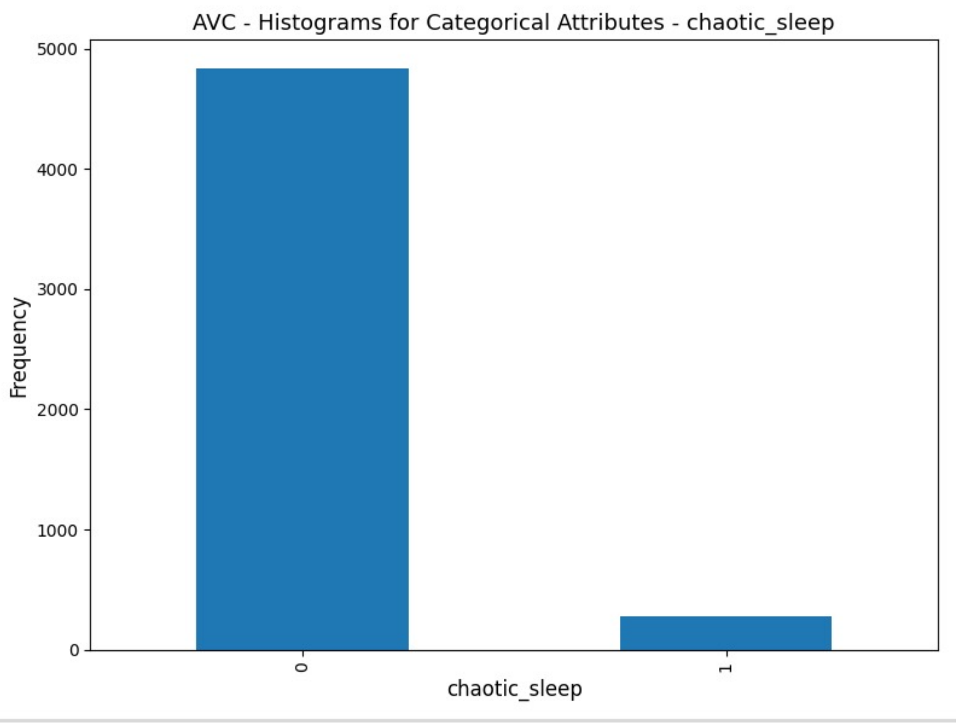


Figure 4: Chaotic Sleep

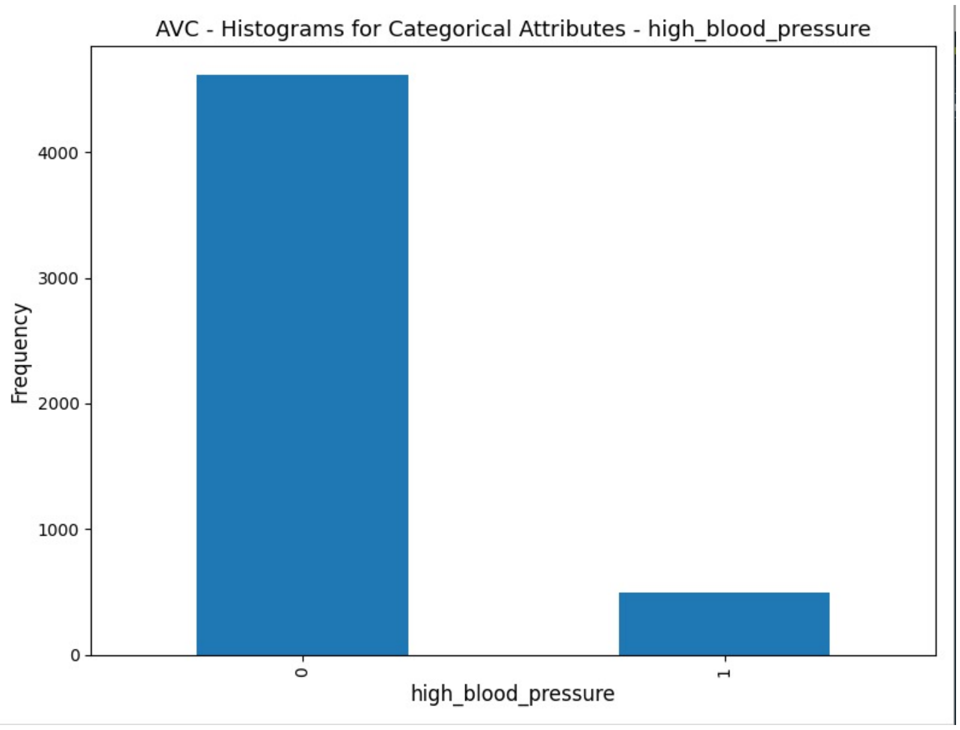


Figure 5: High Blood Pressure

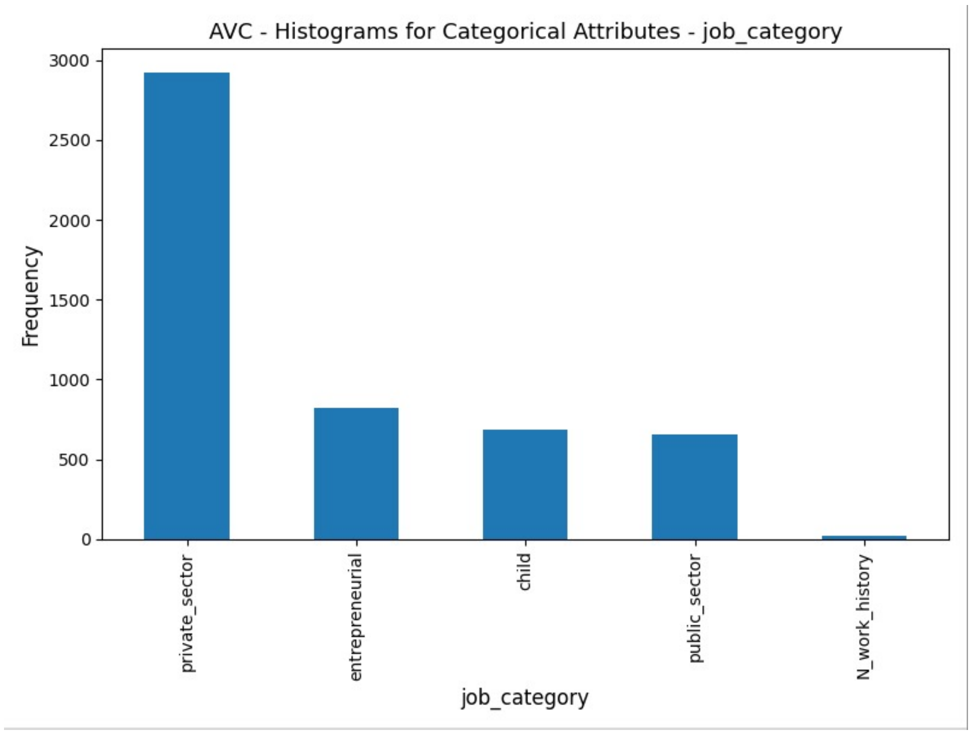


Figure 6: Job Category

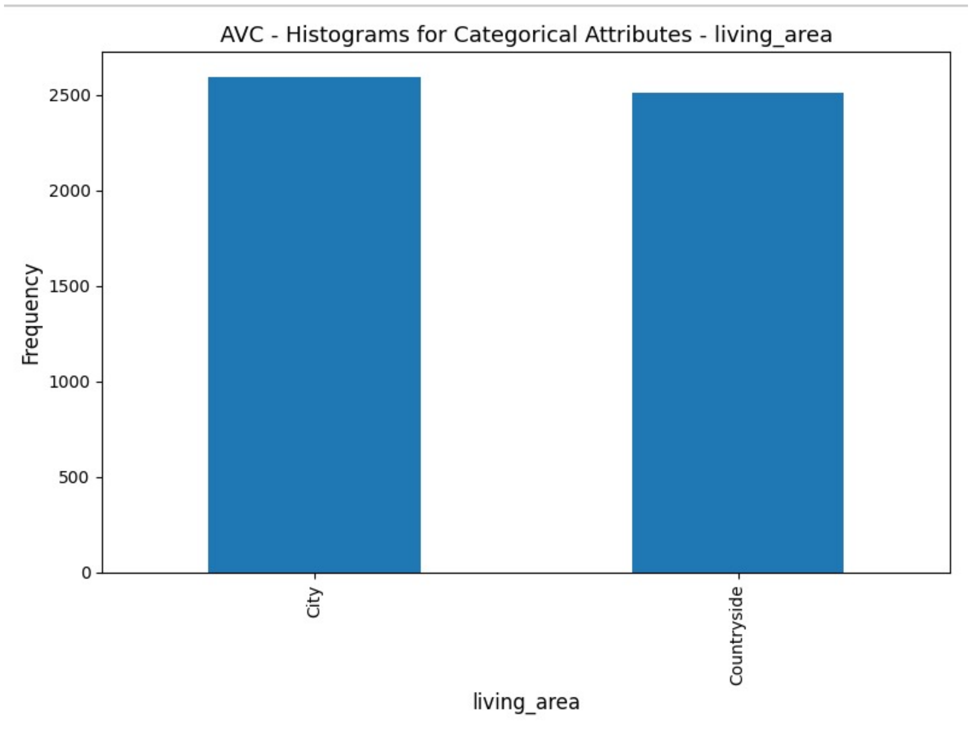


Figure 7: Living Area

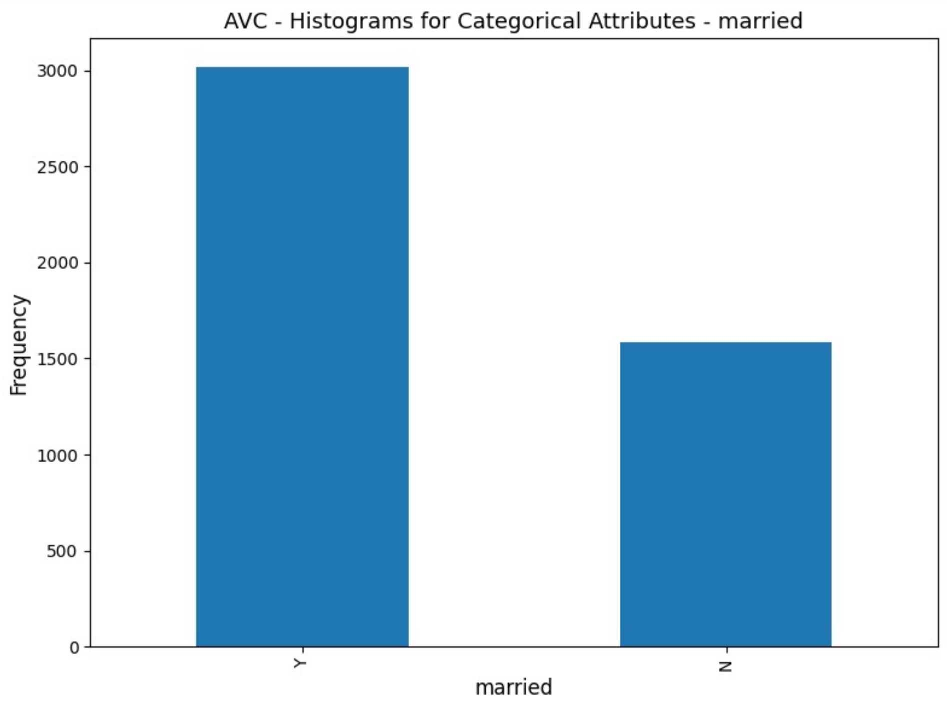


Figure 8: Married

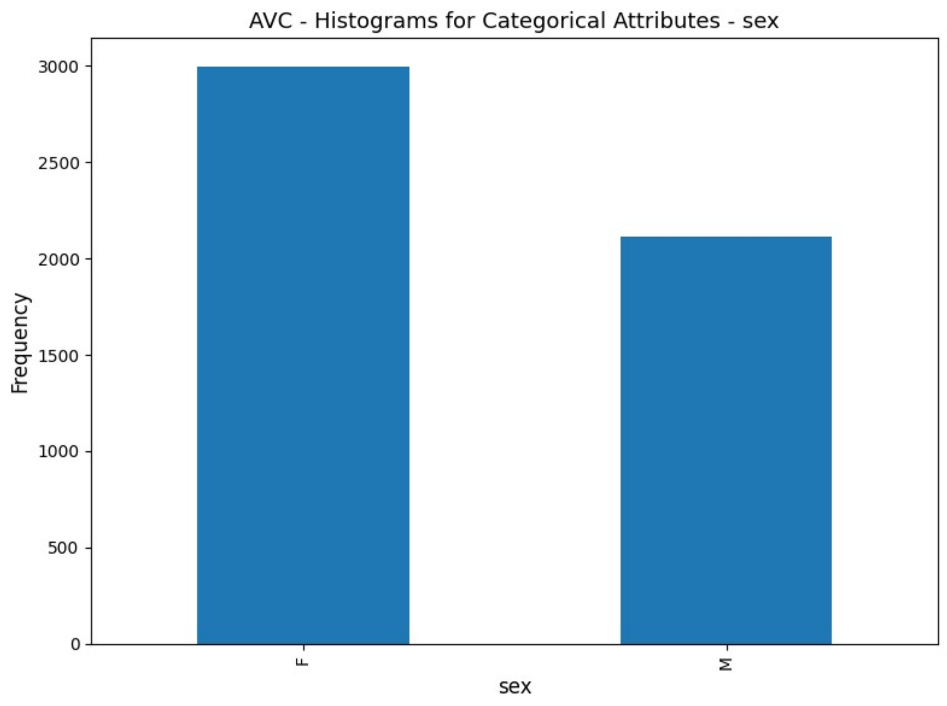


Figure 9: Sex

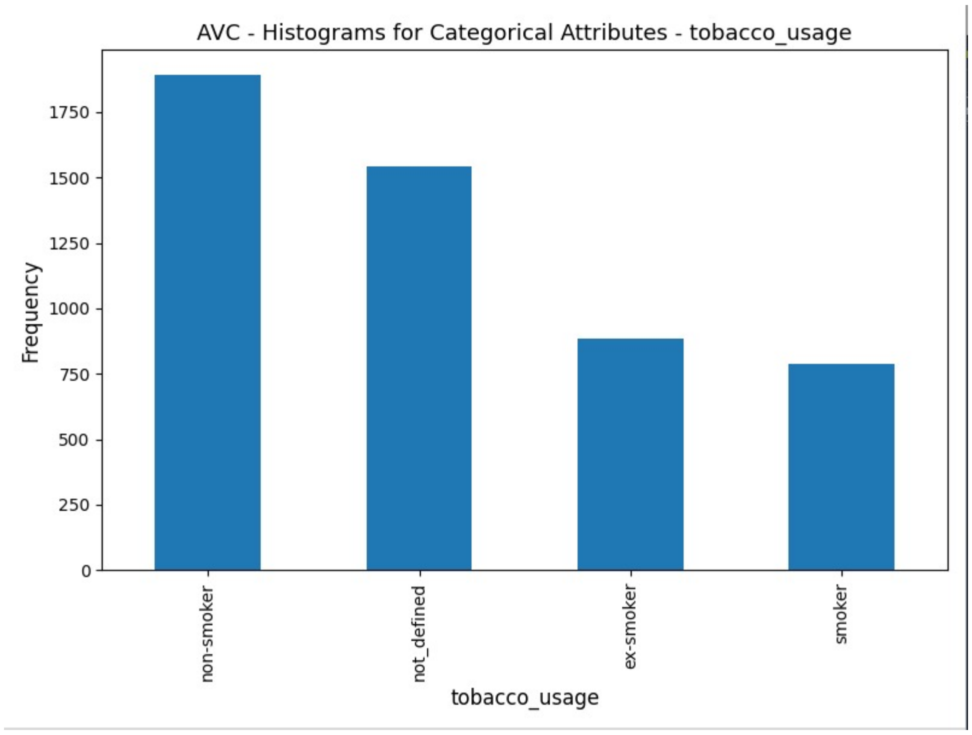


Figure 10: Tobacco Usage

8.2 Salary Dataset

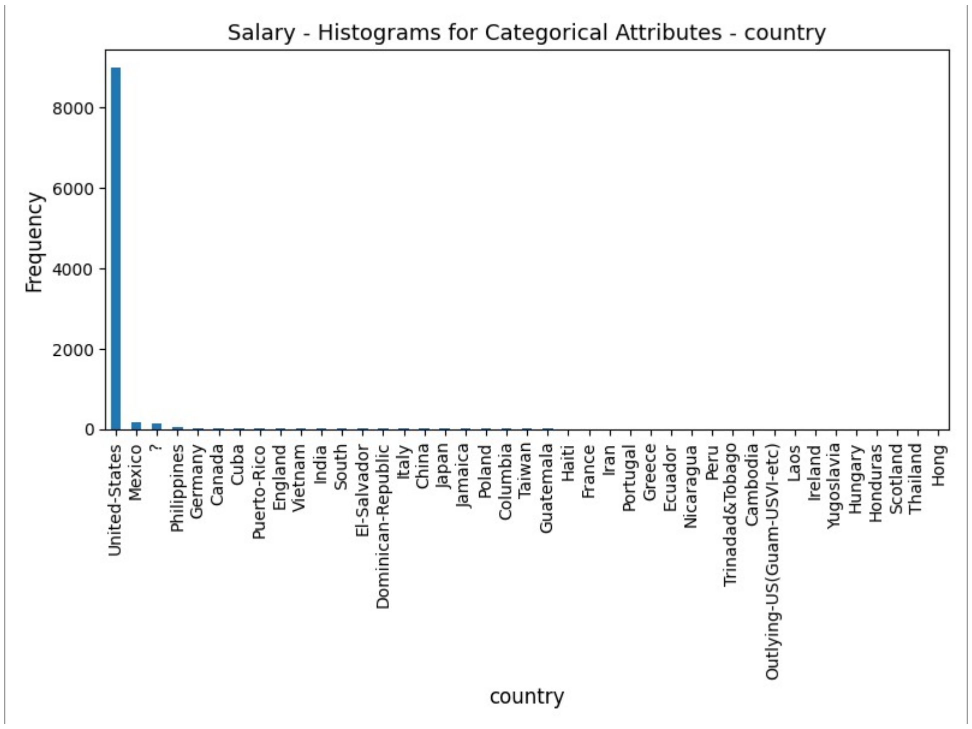


Figure 11: Country

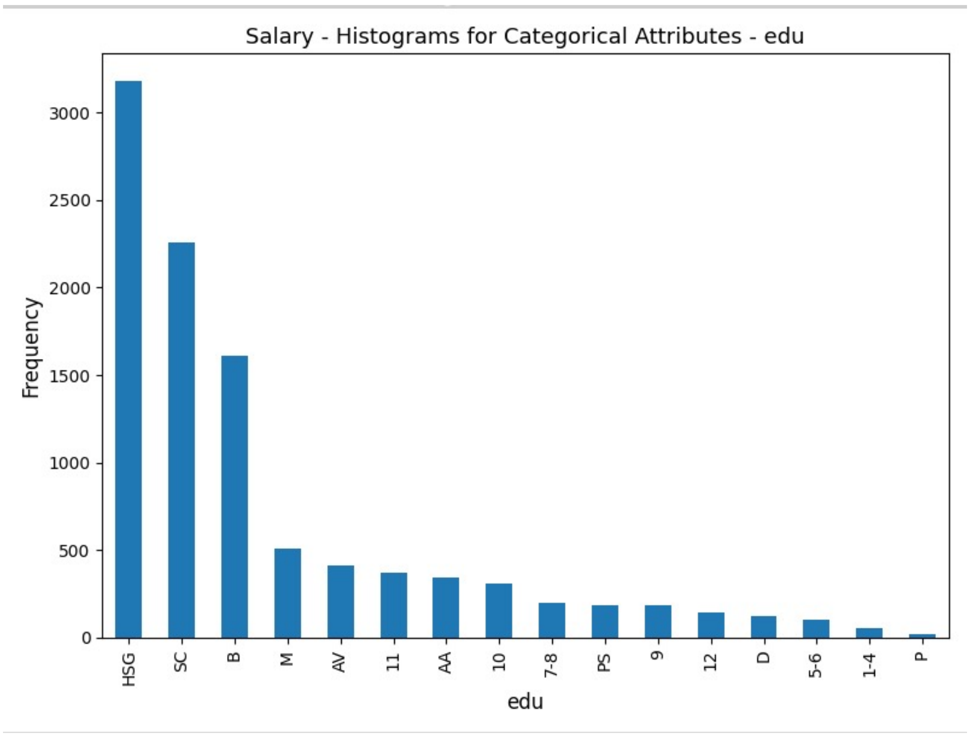


Figure 12: Education

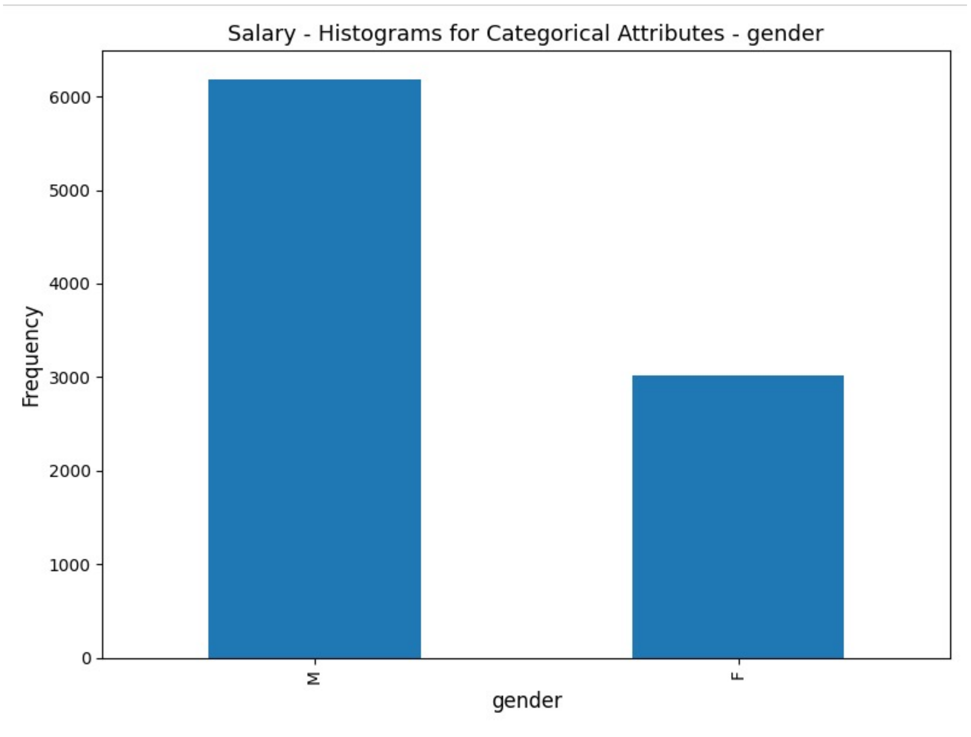


Figure 13: Gender

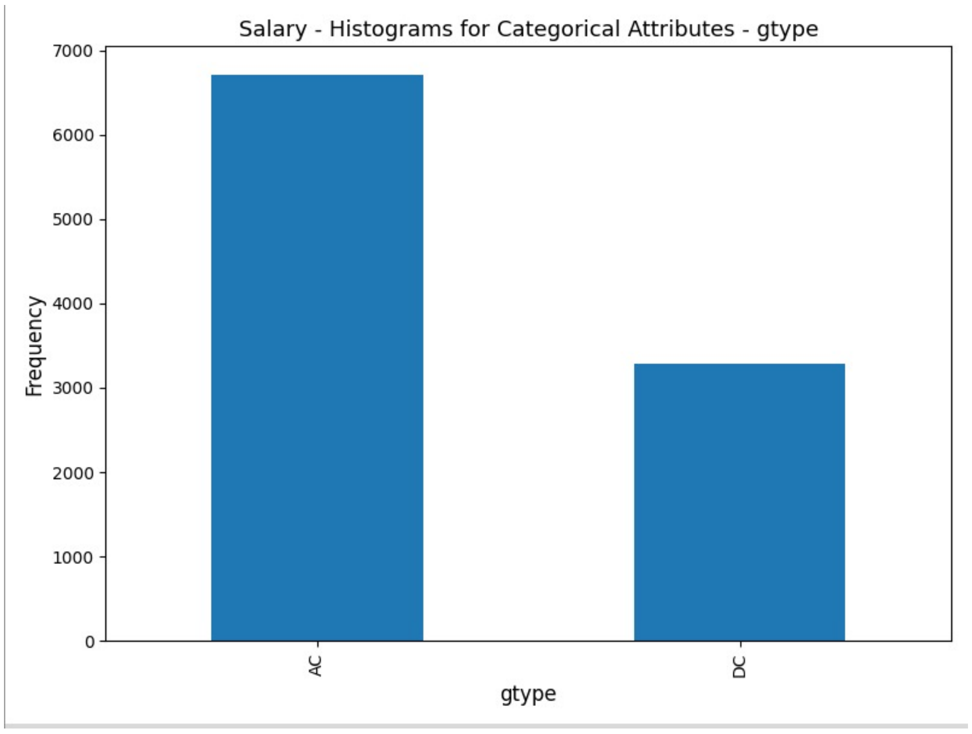


Figure 14: Job Type

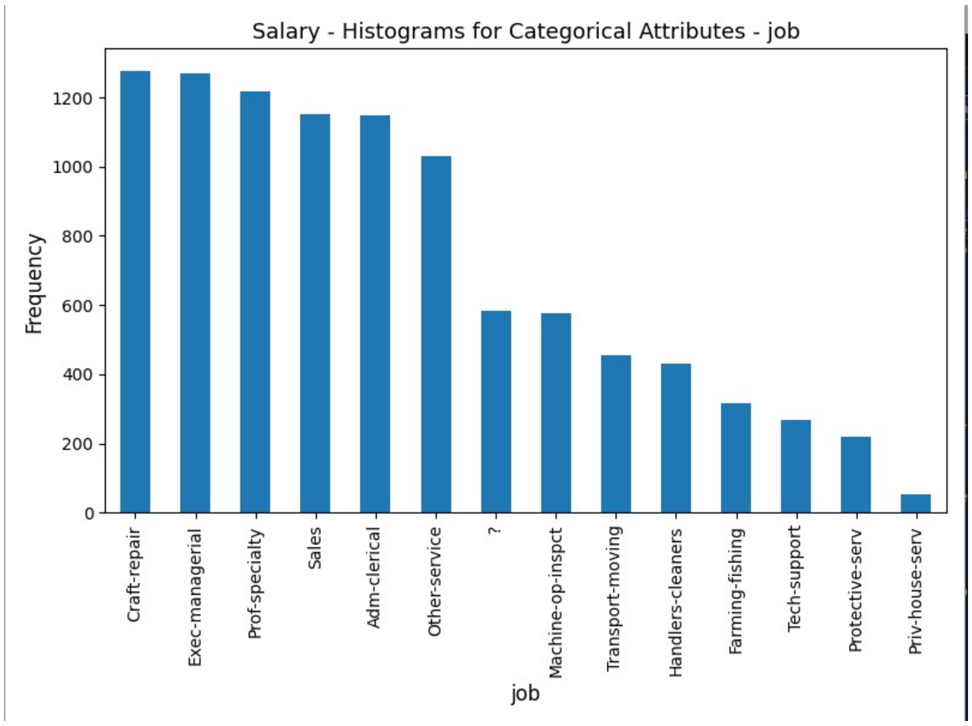


Figure 15: Job

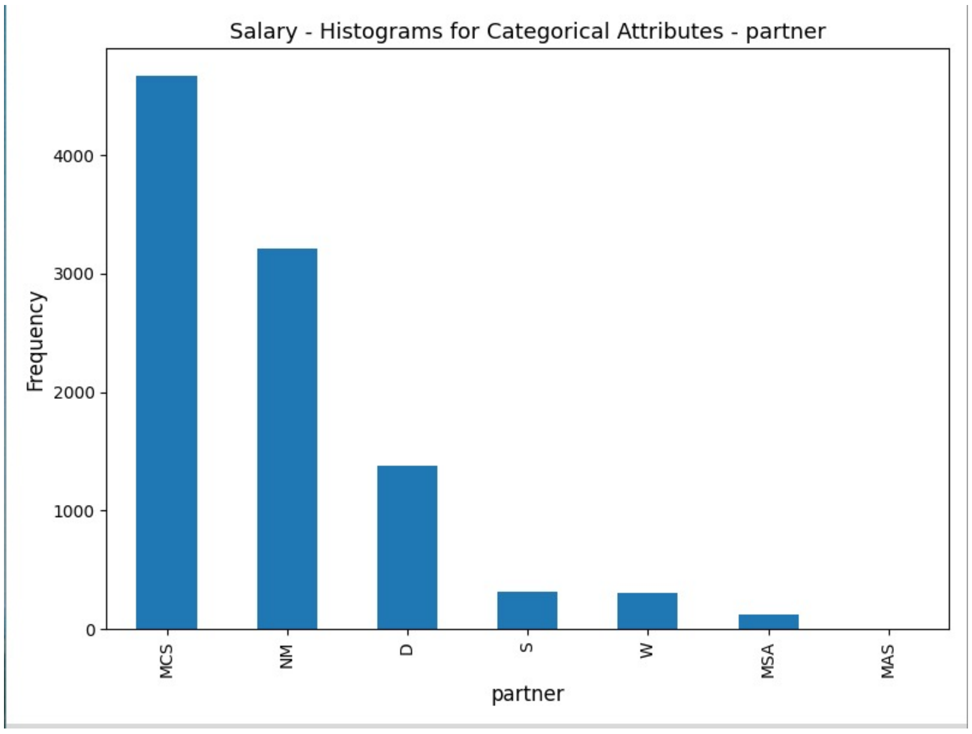


Figure 16: Partner

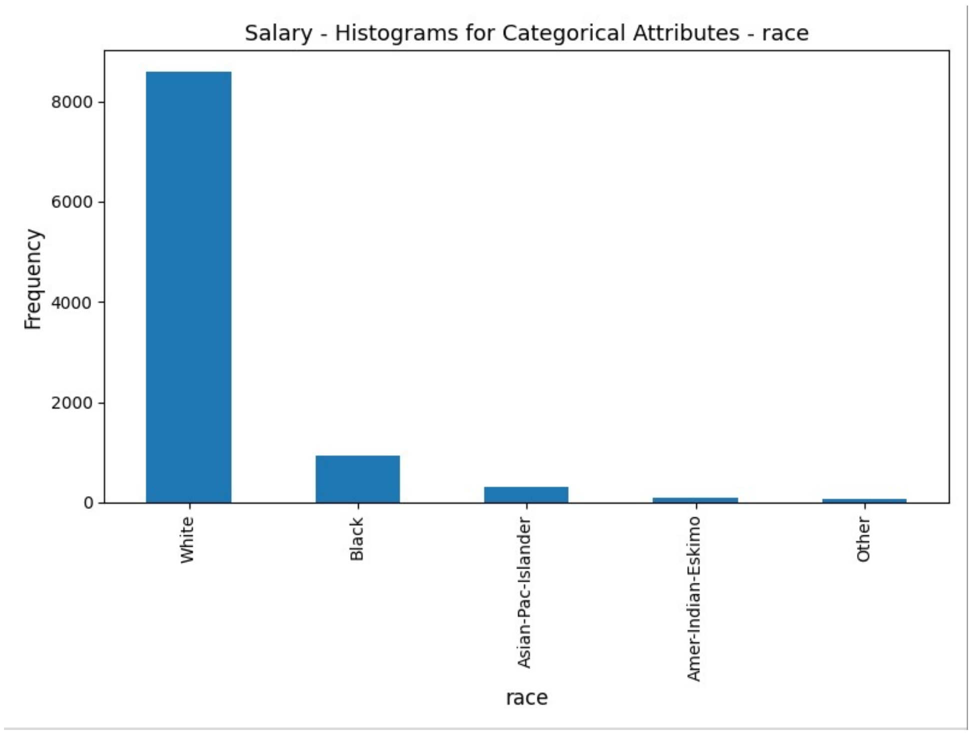


Figure 17: Race

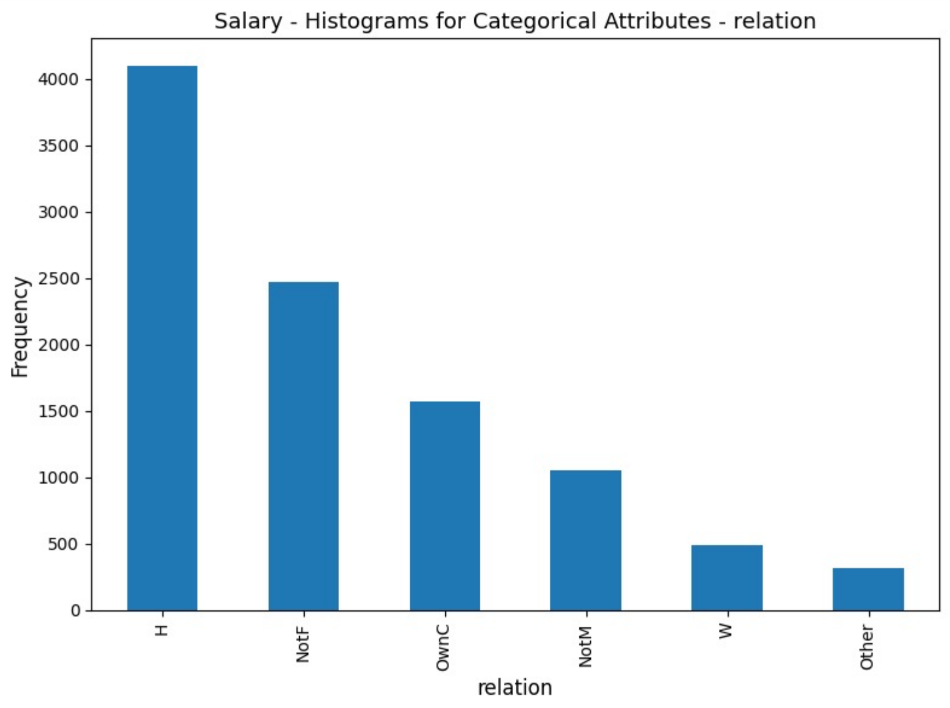


Figure 18: Relation

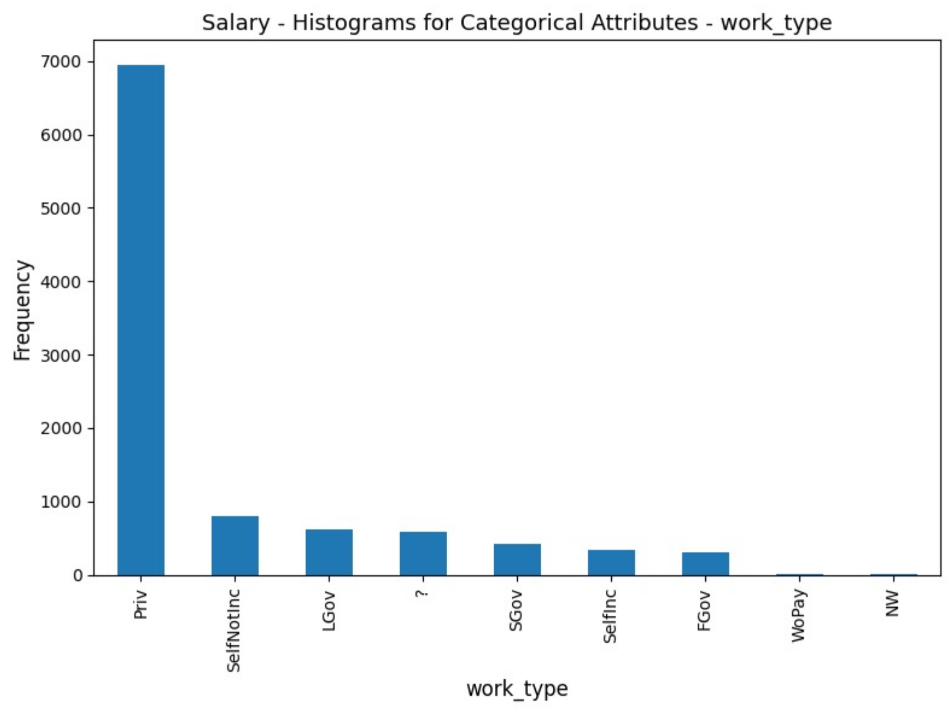


Figure 19: Work Type

9 Class Balance

This section presents the class balance for both the AVC and Salary datasets. The class balance is an important aspect to consider as it influences the evaluation metrics we should focus on.

9.1 AVC Dataset

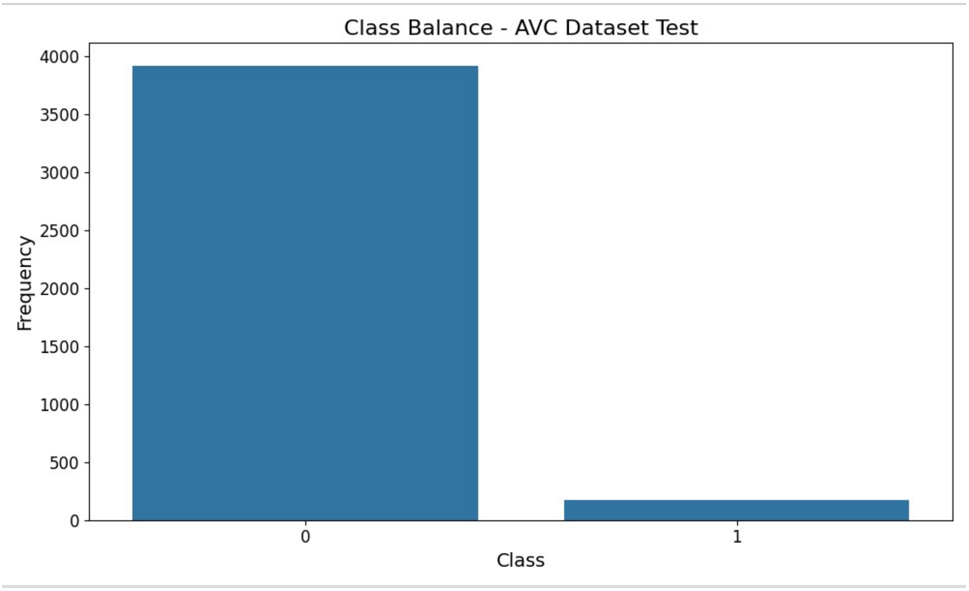


Figure 20: Class Balance - AVC Dataset Test

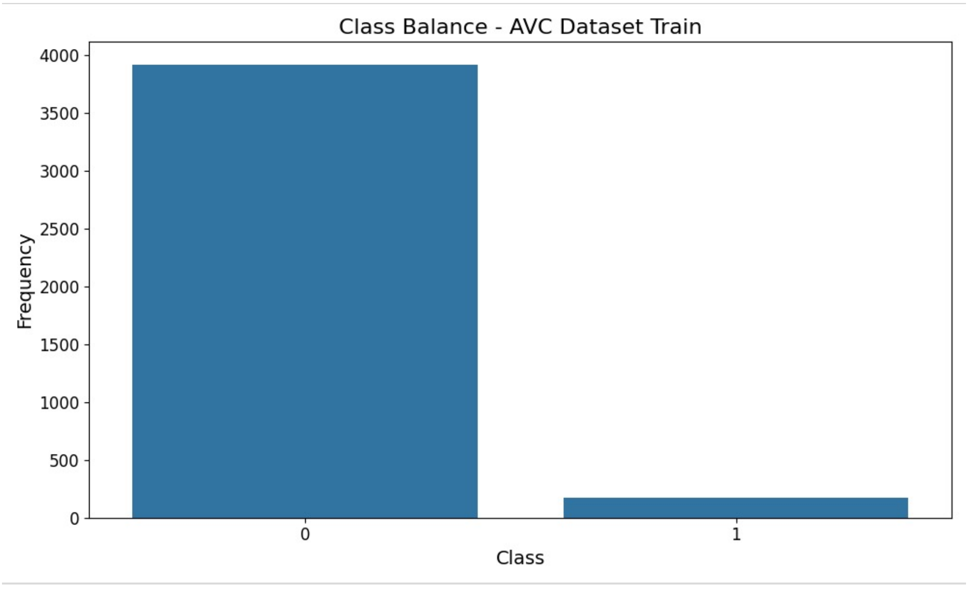


Figure 21: Class Balance - AVC Dataset Train

9.2 Salary Dataset

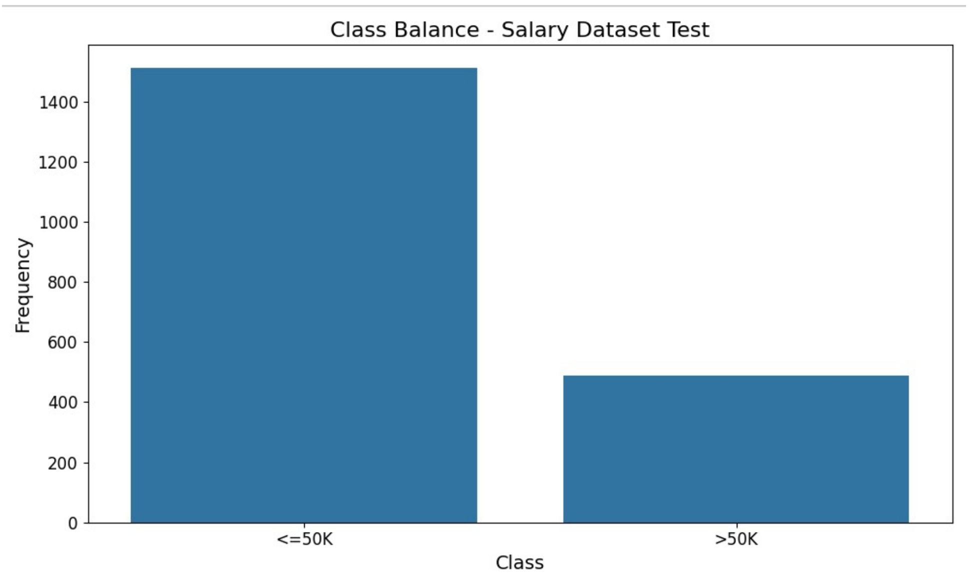


Figure 22: Class Balance - Salary Dataset Test

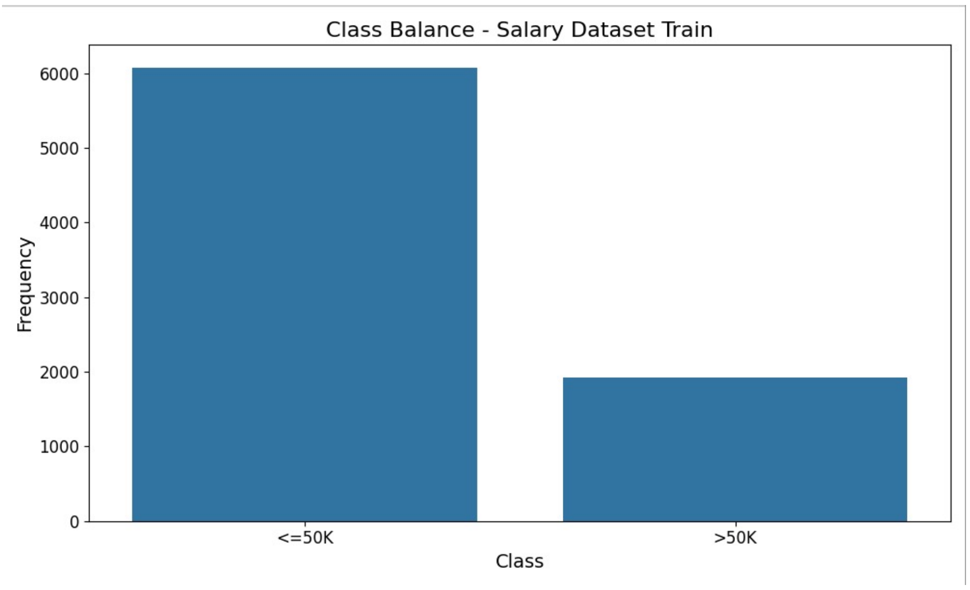


Figure 23: Class Balance - Salary Dataset Train

9.3 Comments

From the class balance plots above, it is evident that both datasets exhibit significant class imbalance. In the AVC dataset, the majority class (no stroke) is much more prevalent than the minority class (stroke). Similarly, in the Salary dataset, the class of individuals earning less than or equal to 50K is more common than the class of individuals earning more than 50K.

Class imbalance can have a substantial impact on model performance and evaluation. Models may become biased towards the majority class, leading to poor performance on the minority class. To address this, it is crucial to focus on evaluation metrics that provide a better understanding of performance on imbalanced datasets. Specifically, we should emphasize:

- **F1 Score**: The harmonic mean of precision and recall, which provides a balance between the two.
- **Precision**: The ability of the classifier to not label a negative sample as positive.
- **Recall**: The ability of the classifier to find all positive samples.

By prioritizing these metrics, we can better assess and compare the performance of different algorithms on these imbalanced datasets.

10 Attribute Correlation

This section presents the correlation matrices for both the AVC and Salary datasets. The correlation matrices are split into numerical and categorical attributes for both train and test sets.

10.1 AVC Dataset

10.1.1 Numerical Attributes

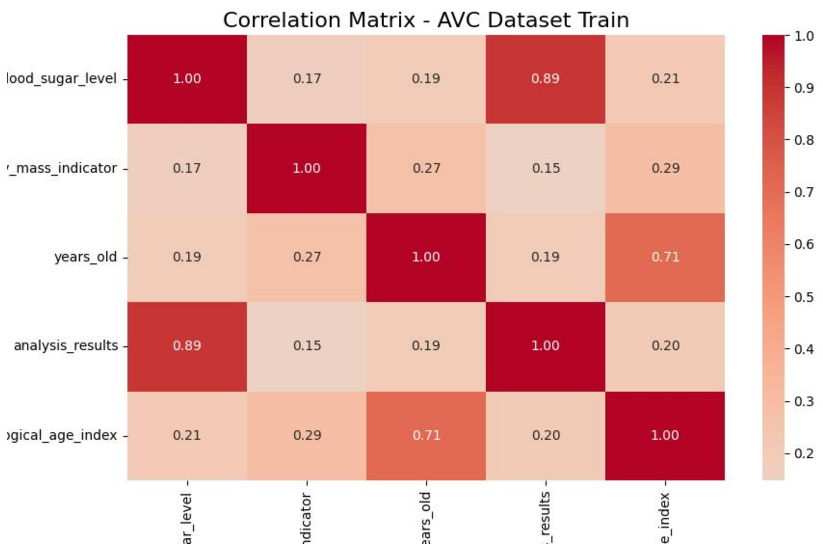


Figure 24: Correlation Matrix - Numerical Attributes - AVC Dataset Train

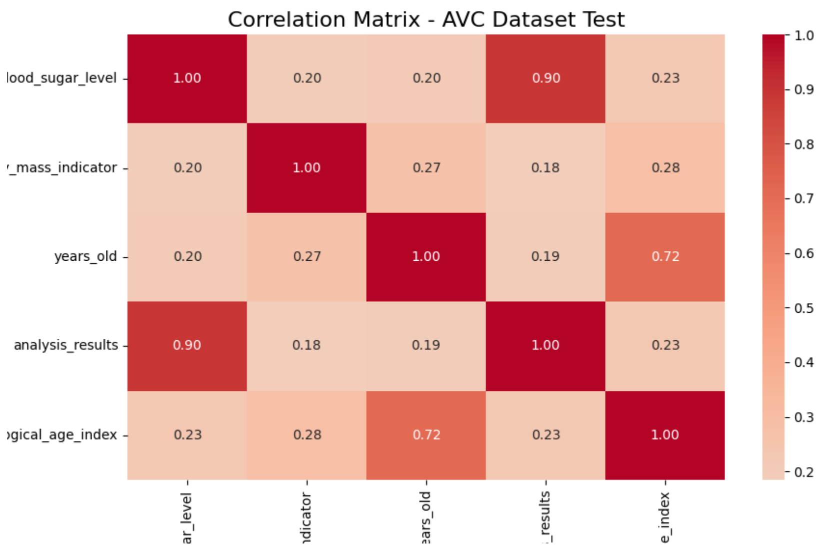


Figure 25: Correlation Matrix - Numerical Attributes - AVC Dataset Test

10.1.2 Categorical Attributes

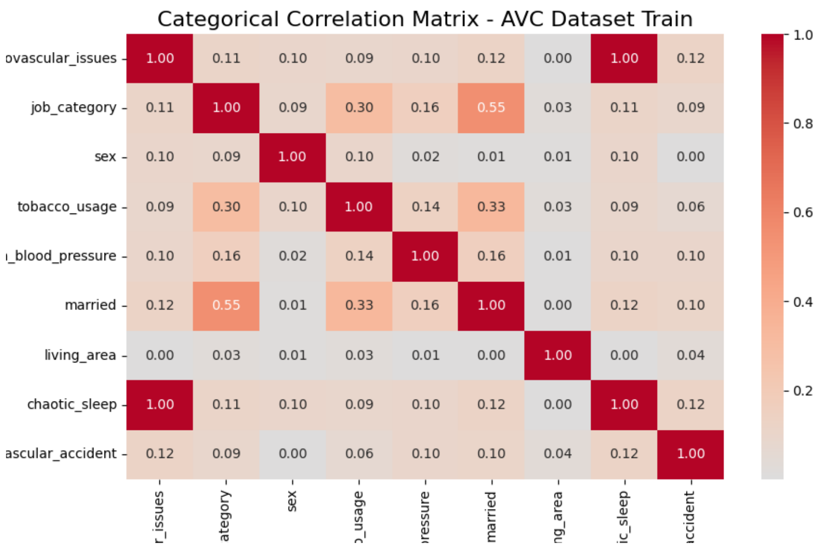


Figure 26: Categorical Correlation Matrix - AVC Dataset Train

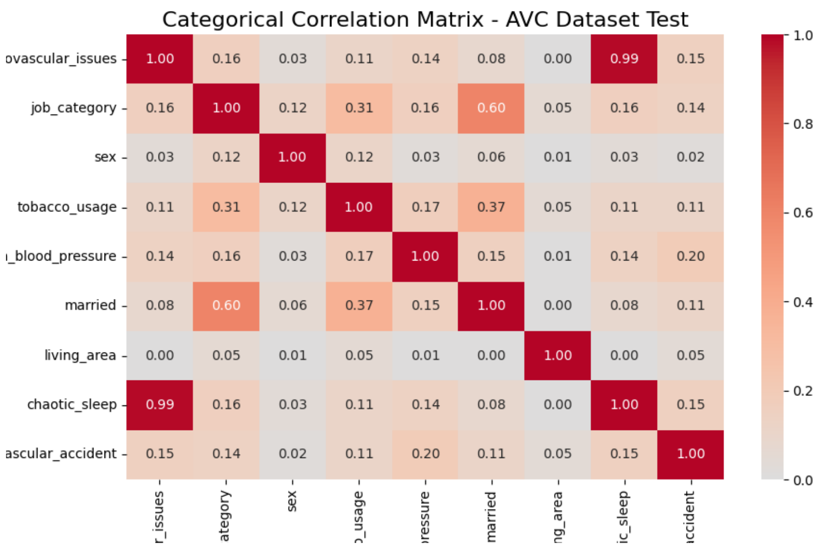


Figure 27: Categorical Correlation Matrix - AVC Dataset Test

10.2 Salary Dataset

10.2.1 Numerical Attributes

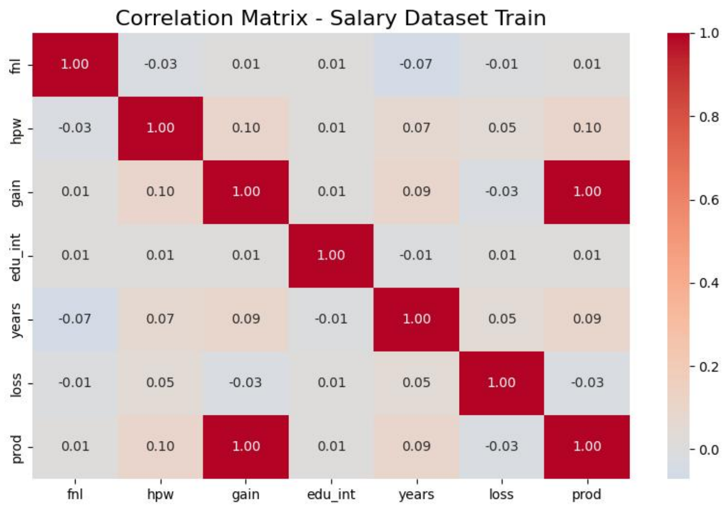


Figure 28: Correlation Matrix - Numerical Attributes - Salary Dataset Train

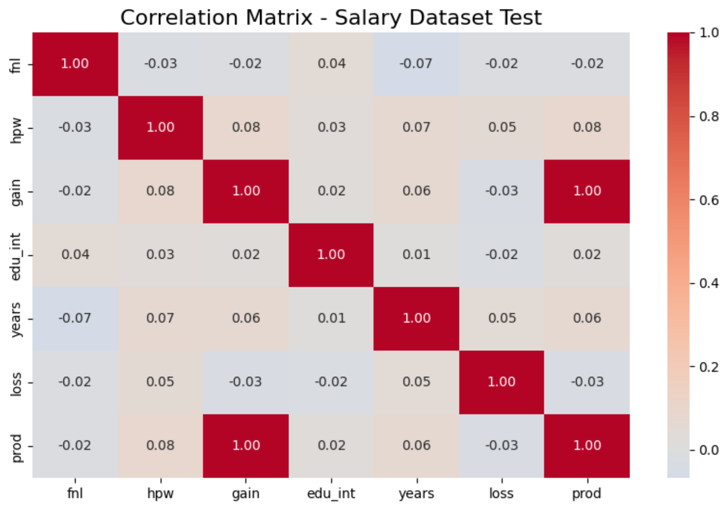
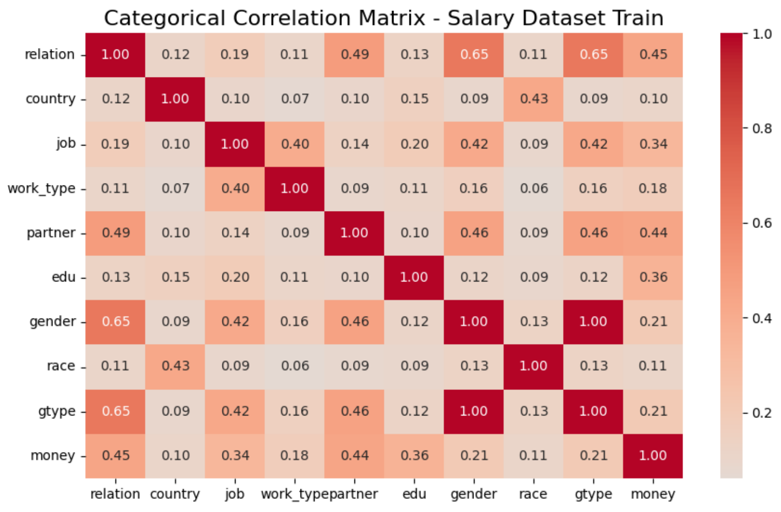


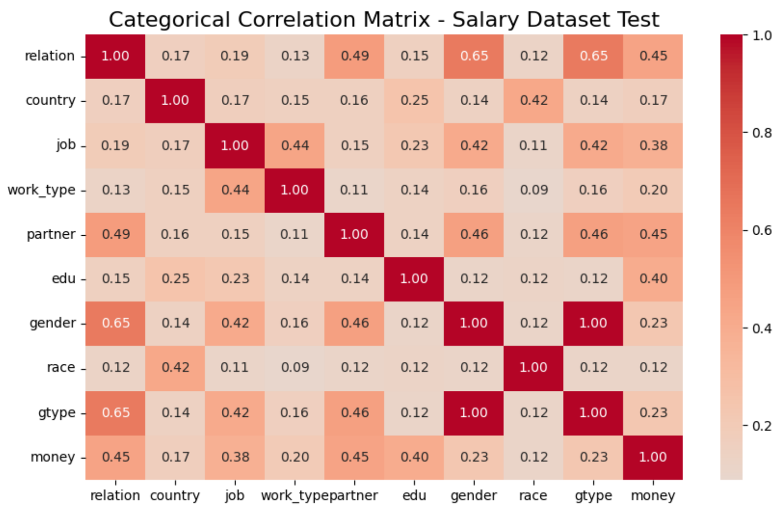
Figure 29: Correlation Matrix - Numerical Attributes - Salary Dataset Test

10.2.2 Categorical Attributes



	relation	country	job	work_type	partner	edu	gender	race	gtype	money
relation	1.00	0.12	0.19	0.11	0.49	0.13	0.65	0.11	0.65	0.45
country	0.12	1.00	0.10	0.07	0.10	0.15	0.09	0.43	0.09	0.10
job	0.19	0.10	1.00	0.40	0.14	0.20	0.42	0.09	0.42	0.34
work_type	0.11	0.07	0.40	1.00	0.09	0.11	0.16	0.06	0.16	0.18
partner	0.49	0.10	0.14	0.09	1.00	0.10	0.46	0.09	0.46	0.44
edu	0.13	0.15	0.20	0.11	0.10	1.00	0.12	0.09	0.12	0.36
gender	0.65	0.09	0.42	0.16	0.46	0.12	1.00	0.13	1.00	0.21
race	0.11	0.43	0.09	0.06	0.09	0.09	0.13	1.00	0.13	0.11
gtype	0.65	0.09	0.42	0.16	0.46	0.12	1.00	0.13	1.00	0.21
money	0.45	0.10	0.34	0.18	0.44	0.36	0.21	0.11	0.21	1.00

Figure 30: Categorical Correlation Matrix - Salary Dataset Train



	relation	country	job	work_type	partner	edu	gender	race	gtype	money
relation	1.00	0.17	0.19	0.13	0.49	0.15	0.65	0.12	0.65	0.45
country	0.17	1.00	0.17	0.15	0.16	0.25	0.14	0.42	0.14	0.17
job	0.19	0.17	1.00	0.44	0.15	0.23	0.42	0.11	0.42	0.38
work_type	0.13	0.15	0.44	1.00	0.11	0.14	0.16	0.09	0.16	0.20
partner	0.49	0.16	0.15	0.11	1.00	0.14	0.46	0.12	0.46	0.45
edu	0.15	0.25	0.23	0.14	0.14	1.00	0.12	0.12	0.12	0.40
gender	0.65	0.14	0.42	0.16	0.46	0.12	1.00	0.12	1.00	0.23
race	0.12	0.42	0.11	0.09	0.12	0.12	0.12	1.00	0.12	0.12
gtype	0.65	0.14	0.42	0.16	0.46	0.12	1.00	0.12	1.00	0.23
money	0.45	0.17	0.38	0.20	0.45	0.40	0.23	0.12	0.23	1.00

Figure 31: Categorical Correlation Matrix - Salary Dataset Test

10.3 Comments

The correlation matrices provide insights into the relationships between different attributes in the datasets.

10.3.1 AVC Dataset

Numerical Attributes From the numerical attribute correlation matrices of the AVC dataset, it is observed that:

- There is a moderate positive correlation between *years_old* and *biological_age_index* in both train and test sets, which indicates that as the age of the person increases, the biological age index also tends to be higher.
- *Analysis_results* show a high correlation with *blood_sugar_level*, but low with other numerical attributes, suggesting that the results of medical analyses are relatively independent of the person's age and other indicators.

Categorical Attributes The categorical correlation matrices for the AVC dataset indicate:

- There are some moderate correlations between categorical variables such as *married* and *job_category*, and *job_category* and *tobacco_usage*.
- These correlations suggest that there might be certain patterns or dependencies between the demographic and lifestyle attributes.
- There is a high correlation between *chaotic_sleep* and *high_blood_pressure*, indicating that individuals with irregular sleep schedules are more likely to have high blood pressure.

10.3.2 Salary Dataset

Numerical Attributes For the Salary dataset, the numerical attribute correlation matrices reveal:

- A strong positive correlation between *gain* and *prod*, indicating that individuals who gain more also tend to have higher produce of capital.
- *Hours per week (hpw)* shows a low correlation with most other numerical attributes, suggesting that the number of hours worked per week is not strongly related to other numerical factors like capital gains or losses.

Categorical Attributes In the categorical correlation matrices for the Salary dataset:

- Moderate correlations are observed between *work_type* and *job*, as well as between *country* and *race*.
- These correlations highlight possible demographic and professional patterns.
- There is also a high correlation between *gtype* and *gender*, indicating that the type of work contract is directly influenced by the gender.

Overall, these correlation matrices help in understanding the interdependencies between attributes, which can be useful in feature selection and in improving the performance of machine learning models.

Eliminating specific variables such as *analysis_results* and *biological_age_index* from the AVC dataset, and prod from the Salary dataset, can enhance the efficiency and performance of our machine learning models. These variables were identified as candidates for elimination due to their high correlation with other attributes, which makes them redundant, and their minimal impact on predictive accuracy. By removing these redundant or less impactful features, we reduce noise, prevent overfitting, and simplify the model, leading to improved generalization and more robust predictive performance. This streamlined approach allows the model to focus on the most significant attributes, enhancing both computational efficiency and model interpretability.

11 Preprocess of Data

Preprocessing data is a crucial step in machine learning. It involves cleaning, transforming, and encoding data to make it suitable for modeling. Below are the steps and functions used in our preprocessing pipeline.

11.1 Imputing Missing Values

Missing values are imputed using the mean for numeric columns and the most frequent value for categorical columns.

```
def impute_missing_values(df, nmr_cols, cat_cols, fit, imp):
    df_nmr = df[nmr_cols]
    df_cat = df[cat_cols]
    df.replace('?', np.nan, inplace=True)
    if fit:
        # Fit imputer if on training data
        imp = SimpleImputer(strategy='mean')
        df_nmr_imputed = pd.DataFrame(imp.fit_transform(df_nmr), cols=nmr_cols)
        imp = SimpleImputer(strategy='most_frequent')
        df_cat_imputed = pd.DataFrame(imp.fit_transform(df_cat), cols=cat_cols)
    else:
        # Use the fitted imputer from training data
        df_nmr_imputed = pd.DataFrame(imp['nmr'].transform(df_nmr), cols=nmr_cols)
        df_cat_imputed = pd.DataFrame(imp['cat'].transform(df_cat), cols=cat_cols)

    df_imputed = pd.concat([df_nmr_imputed, df_cat_imputed], axis=1)
    return df_imputed, imputers
```

11.2 Handling Extreme Values

Extreme values are handled using the Interquartile Range (IQR) method and then imputed.

```
def handle_extreme_values(df, nmr_cols):
    # Handling extreme values using IQR method
    Q1 = df[nmr_cols].quantile(0.25)
    Q3 = df[nmr_cols].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR

    # Replace extreme values with NaN
    for col in nmr_cols:
        df[col] = np.where((df[col] < lower_bound[col]) |
                           (df[col] > upper_bound[col]),
                           np.nan, df[col])

    # Impute missing values for the added NaNs
    imputer = SimpleImputer(strategy='mean')
    df[nmr_cols] = imputer.fit_transform(df[nmr_cols])
    return df
```

11.3 Removing Highly Correlated Attributes

Attributes with high or medium high correlation (above 0.5) are removed to prevent multicollinearity. Correlation classes:
-0 to 0.3: low correlation
-0.3 to 0.7: medium correlation
-0.7 to 1: high correlation

This resulted in the following eliminations: **analysis_results** and **biological_age_index** from the AVC dataset, and **prod** from the Salary dataset.

```
def remove_redundant_attributes(df, nmr_cols, threshold=0.9):
    numeric_df = df[nmr_cols]

    # Get upper triangle of correlation matrix
    corr_mat = numeric_df.corr().abs()
    upper = corr_mat.where(np.triu(np.ones(corr_mat.shape), k=1).astype(bool))

    # Drop everything higher than threshold
    to_drop = [col for col in upper.cols if any(upper[col] > threshold)]
    df_reduced = df.drop(cols=to_drop)
    return df_reduced, to_drop
```

11.4 Standardizing Numerical Attributes

Standardizing ensures that the numerical attributes have a mean of 0 and a standard deviation of 1.

```
def standardize_data(df, nmr_cols, method='standard', fit=True, scaler=None):
    if fit:
        if method == 'standard':
            scaler = StandardScaler()
        elif method == 'minmax':
            scaler = MinMaxScaler()
        elif method == 'robust':
            scaler = RobustScaler()
        df[nmr_cols] = scaler.fit_transform(df[nmr_cols])
        return df, scaler
    else:
        df[nmr_cols] = scaler.transform(df[nmr_cols])
        return df
```

11.5 Encoding Categorical Variables

Categorical variables are encoded using OneHotEncoder, and the target variable is encoded using LabelEncoder.

```
def encode_categorical(df, trg_col, enc=None):
    df = df.copy()
    cat_cols = df.select_dtypes(include=['object']).cols.tolist()

    # Remove target column from categorical columns
    if trg_col in categorical_cols:
        cat_cols.remove(trg_col)

    # Encode target column
    label_enc = LabelEncoder()
    df[trg_col] = label_enc.fit_transform(df[trg_col])

    # Encode categorical columns
    enc = OneHotEncoder(sparse_output=False, drop='first', handle_unknown='ignore')
    df_onehot = pd.DataFrame(enc.fit_transform(df[categorical_cols]),
                             cols=enc.get_feature_names_out(categorical_cols))

    df = pd.concat([df, df_onehot], axis=1)
    df.drop(categorical_cols, axis=1, inplace=True)

    return df, label_enc, enc
```

11.6 Wrapper Function for Preprocessing

The wrapper function applies the above steps to each dataset (training and test) and saves the processed data.

```
def preprocess_data_wrapper():
    for name, df in datasets.items():
        if ’train’ in name:
            df_reduced, dropped_cols = rm_redundant(..., threshold=0.5)
            nmr_cols_after_drop = [col for col in nmr_cols if col not in dropped]

            df_imputed, imputer = inpute_missing(...)
            fitted_imputers[name] = imputer

            df_outliers_handled = handle_extreme_values(...)

            df_standardized, scaler = standardize_data(...)
            fitted_scalers[name] = scaler

            df_encoded, label_enc, onehot_enc = encode_categorical(...)
            fitted_encs[name] = (label_enc, onehot_enc)

            processed_datasets[name] = df_encoded
        else:
            dropped = [col for col in nmr_cols if col not in proc_df.cols]
            nmr_cols_after_drop = [col for col in nmr_cols if col not in dropped]

            imputer = fitted_imputers[train_name]
            scaler = fitted_scalers[train_name]

            df_imputed, _ = impute_missing_values(...)
            df_outliers_handled = handle_extreme_values(...)
            df_outliers_imputed, _ = impute_missing_values(...)
            df_standardized = standardize_data(...)
            df_encoded, _, _ = encode_categorical(...)

            train_cols = processed_datasets[train_name].cols
            for col in train_cols:
                if col not in df_encoded.cols:
                    df_encoded[col] = 0
            df_encoded = df_encoded[train_cols]

            processed_datasets[name] = df_encoded

    X_avc_train, T_avc_train, _, _ = preprocess_data(...)
    X_avc_test, T_avc_test, _, _ = preprocess_data(...)
    X_salary_train, T_salary_train, _, _ = preprocess_data(...)
    X_salary_test, T_salary_test, _, _ = preprocess_data(...)
    return return_tuple_avc, return_tuple_salary
    def preprocess_data(df, target_column, enc=None):
        # Preprocess the data by encoding cat variables and separating features
        df_enc, label_enc, enc = encode_categorical(df, target_column, enc)
        # Separate features
        X = df_enc.drop(columns=[target_column]).values
        # Separate target
        T = df_enc[target_column].values
        return X, T, label_enc, enc
```

12 Logistic Regression

12.1 Manual Implementation

In this section, we implement logistic regression from scratch. The steps involved are as follows:

- **Logistic Function**: Apply the logistic (sigmoid) function to transform input values into probabilities.
- **Negative Log-Likelihood (NLL)**: Compute the NLL, which measures how well the predicted probabilities match the true labels.
- **Accuracy**: Calculate the accuracy of the predictions by comparing them to the true labels.
- **Prediction**: Make predictions using the logistic regression model.
- **Training and Evaluation**: Train the logistic regression model using gradient descent and evaluate its performance.

```
function logistic(x):
    # Sigmoid function
    return 1 / (1 + exp(-x))

function nll(Y, T):
    # Negative Log-Likelihood
    N = number of samples
    return -sum(T * log(Y) + (1 - T) * log(1 - Y)) / N
```

```
function accuracy(Y, T):
    # Accuracy calculation
    N = number of samples
    acc = 0

    # Compare predictions to true labels
    for i in range(N):
        if (Y[i] >= 0.5 and T[i] == 1) or (Y[i] < 0.5 and T[i] == 0):
            acc += 1
    return acc / N

function predict_logistic(X, w):
    # Make predictions using logistic regression model
    return logistic(dot(X, w))

function train_and_eval_logistic(X_train, T_train, X_test, T_test, lr, epochs_no):
    N, D = shape of X_train
    w = random weights
    train_acc, test_acc = []
    train_nll, test_nll = []

    for epoch in range(epochs_no):
        # Make predictions
        Y_train = predict_logistic(X_train, w)
        Y_test = predict_logistic(X_test, w)

        # Calculate metrics
        train_acc.append(accuracy(Y_train, T_train))
        test_acc.append(accuracy(Y_test, T_test))
        train_nll.append(nll(Y_train, T_train))
        test_nll.append(nll(Y_test, T_test))

        # Update weights using gradient descent
        w = w - lr * dot(transpose(X_train), (Y_train - T_train)) / N

    return w, train_nll, test_nll, train_acc, test_acc
```

12.2 Scikit-learn Implementation

In this section, we use the `scikit-learn` library to implement logistic regression. The steps involved are:

- **Model Initialization**: Initialize the logistic regression model with a maximum of 1000 iterations.
- **Model Fitting**: Fit the model to the training data.
- **Prediction**: Make predictions on the training and test sets.
- **Evaluation**: Calculate accuracy and NLL for the training and test sets.

```
function train_and_eval_sklearn_logistic(X_train, T_train, X_test, T_test):
    # Initialize logistic regression model
    model = LogisticRegression(max_iter=1000)

    # Fit the model
    model.fit(X_train, T_train)

    # Make predictions
    Y_train = model.predict_proba(X_train)[:, 1]
    Y_test = model.predict_proba(X_test)[:, 1]

    # Calculate metrics
    train_acc = accuracy_score(T_train, (Y_train >= 0.5).astype(int))
    test_acc = accuracy_score(T_test, (Y_test >= 0.5).astype(int))
    train_nll = log_loss(T_train, Y_train)
    test_nll = log_loss(T_test, Y_test)

    return model, train_nll, test_nll, train_acc, test_acc
```

12.3 Particularities

In this section, we document the specific approach and settings used for our logistic regression algorithm.

12.3.1 Categorical Attribute Encoding

For encoding categorical attributes, we employed the following approach:

- **Label Encoding**: The target variable was encoded using Label Encoding, which converts categorical labels into numerical values.
- **One-Hot Encoding**: Other categorical attributes were encoded using One-Hot Encoding. This method creates binary columns for each category, representing the presence or absence of a specific category in the data.

12.3.2 Optimization Algorithm Settings

We utilized gradient descent as the optimization algorithm with the following settings:

- **Learning Rate**: A learning rate of 0.1 was chosen to control the step size during the update of the weights. This value was selected based on preliminary experiments to balance convergence speed and stability.
- **Epochs**: The number of epochs was set to 1000 to allow the model to converge to an optimal solution. This value was chosen based on the convergence behavior observed during training.

13 MLP Implementation

13.1 Activation Functions

The following activation functions are used in our MLP implementation:

- **Sigmoid**: $\sigma(x) = \frac{1}{1+e^{-x}}$
- **ReLU (Rectified Linear Unit)**: $\text{ReLU}(x) = \max(0, x)$
- **Tanh**: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- **Leaky ReLU**: $\text{Leaky ReLU}(x) = \max(\alpha x, x)$
- **ELU (Exponential Linear Unit)**: $\text{ELU}(x) = x$ if $x > 0$, $\alpha(e^x - 1)$ if $x \leq 0$

13.2 Manual Implementation

In this section, we manually implement an MLP with the following components:

- **Activation Functions**: Sigmoid, ReLU, Tanh, Leaky ReLU, and ELU
- **Layers**: Linear layer and Dropout layer
- **Loss Function**: Cross-entropy loss
- **Optimizer**: SGD and Adam optimizers

```
class Linear:
    def __init__(input_dim, output_dim, l2_reg=0.01):
        Initialize weights and biases

    def forward(x):
        return dot(x, weight) + bias
    def backward(x, dldy):
        Compute gradients for weights and biases

    def update(lr):
        Update weights and biases using gradients

class Dropout:
    def __init__(rate=0.5):
        Set dropout rate

    def forward(x, train=True):
        Apply dropout during training

    def backward(x, dldy):
        Return dldy * mask
```

```
class FeedForwardNetwork:
    def __init__(layers):
        Initialize network with layers

    def forward(x, train=True):
        Forward pass through all layers
    def backward(dldy):
        Backward pass through all layers

    def update(*args):
        Update parameters for all layers

function train_and_evaluate_manual_mlp(X_train, T_train, X_test, T_test,
                                       input_size, hidden_size, output_size,
                                       epochs, learning_rate, l2_reg, batch_size):
    Initialize MLP with Linear, Activation, Dropout layers
    Initialize CrossEntropy loss and SGD optimizer
    for epoch in range(epochs):
        Shuffle and batch data
        for each batch:
            Forward pass
            Compute loss
            Backward pass
            Update parameters
        Evaluate on training and test sets
    return mlp, train_acc_list, test_acc_list, train_loss_list, test_loss_list
```

13.3 Scikit-learn Implementation

In this section, we use the `scikit-learn` library to implement an MLP. The steps involved are:

- **Model Initialization**: Initialize the MLP classifier with specified parameters.
- **Model Fitting**: Fit the model to the training data.
- **Prediction**: Make predictions on the training and test sets.
- **Evaluation**: Calculate accuracy for the training and test sets.

```
function train_and_evaluate_sklearn_mlp(X_train, T_train, X_test, T_test,
                                        hidden_layer_sizes, max_iter,
                                        learning_rate_init, alpha):
    # Initialize the MLP classifier with the specified parameters
    mlp = MLPClassifier(hidden_layer_sizes=hidden_layer_sizes,
                        max_iter=max_iter, learning_rate_init=learning_rate_init,
                        alpha=alpha, random_state=1)

    # Fit the MLP classifier to the training data
    mlp.fit(X_train, T_train)

    # Make predictions on the training and test sets
    train_predictions = mlp.predict(X_train)
    test_predictions = mlp.predict(X_test)

    # Compute accuracy for training and test sets
    train_accuracy = accuracy_score(T_train, train_predictions)
    test_accuracy = accuracy_score(T_test, test_predictions)

    return train_accuracy, test_accuracy, mlp
```

Using `scikit-learn`, we leverage pre-built functions for model training and evaluation, simplifying the implementation process while maintaining performance.

13.4 Optimizer Usage

In our MLP implementation, we utilized two different optimizers to update the weights and biases of the network during training: Stochastic Gradient Descent (SGD) and Adaptive Moment Estimation. After extensive testing, we found that the SGD optimizer resulted in better overall performance.

13.4.1 SGD Optimizer

The SGD optimizer updates the weights and biases of the linear layers using the gradient of the loss function with respect to each parameter.

```
class SGDOptimizer:
    def __init__(self, layers, learning_rate=0.001):
        Initialize learning rate and select linear layers

    def update():
        for each linear layer:
            Update weight using gradient descent
            Update bias using gradient descent
```

The SGD optimizer provided better results in our experiments, possibly due to its simplicity and effectiveness in handling the specific characteristics of our datasets.

13.4.2 Adam Optimizer

The Adam optimizer is an advanced optimization technique that combines the advantages of both the AdaGrad and RMSProp algorithms. It uses adaptive learning rates for each parameter and maintains running averages of the first and second moments of the gradients. The pseudocode for the Adam optimizer is as follows:

```
class AdamOptimizer:
    def __init__(self, layers, learning_rate, beta1, beta2, epsilon):
        Initialize parameters and select linear layers

    def update():
        Increment timestep
        for each linear layer:
            Update first moment estimate (m)
            Update second moment estimate (v)
            Compute bias-corrected first moment estimate (m_hat)
            Compute bias-corrected second moment estimate (v_hat)
            Update weight using Adam update rule
            Update bias using Adam update rule
```

Although the Adam optimizer is generally known for its robust performance across various tasks, in our specific case, the SGD optimizer provided better results. This may be due to the nature of our data and the specific architecture of our MLP, where the more straightforward approach of SGD was more effective.

13.5 Particularities

In this section, we document the hyperparameterization and specific configurations used for the Multi-Layer Perceptron (MLP) neural network.

13.5.1 Architecture

For the architecture of the MLP, the following configurations were used:

- **Number and Size of Layers**:
  - **AVC Dataset (Manual Implementation)**:
    - Input Layer: Size equal to the number of features in the AVC training set.
    - Hidden Layer: 128 neurons or two layers with 100/50 neurons.
    - Output Layer: 2 neurons (binary classification).
  - **Salary Dataset (Manual Implementation)**:
    - Input Layer: Size equal to the number of features in the Salary training set.
    - Hidden Layer: 128 neurons or two layers with 100/50 neurons.
    - Output Layer: 2 neurons (binary classification).

  - **Scikit-learn Implementation**:
    * Hidden Layers: (100, 50) neurons for each dataset.

- **Activation Functions**:
  - ELU (Exponential Linear Unit) was used in the manual implementation proven to be effective in our experiments.
  - Scikit-learn MLP uses ReLU (Rectified Linear Unit) by default.

13.5.2 Optimizer Configuration

The optimizer settings were as follows:

- **Type of Optimizer**:
  - Manual Implementation: Stochastic Gradient Descent (SGD) optimizer was used.
  - Scikit-learn Implementation: The default optimizer in scikit-learn MLPClassifier.
- **Learning Rate**:
  - Manual Implementation: 0.00002
  - Scikit-learn Implementation: 0.01
- **Number of Epochs**: 1000 epochs were used for training.
- **Batch Size**: A batch size of 16/32/64 was used for training.

13.5.3 Regularization Methods

To prevent overfitting and ensure the model generalizes well, the following regularization methods were applied:

- **Early Stopping**: Tested and proven to result in unpredictable loss variation.
- **L2 Regularization**:
  - Manual Implementation: A regularization coefficient (l2_reg) of 0.001 was used. Weight is set as random.randn(input_dim, output_dim) * sqrt(2 / input_dim)
  - Scikit-learn Implementation: An alpha value of 0.0001 for L2 regularization was used.
- **Dropout**: A dropout rate of 0.5 was applied in the manual implementation to prevent overfitting.

These configurations were selected to balance model complexity, training efficiency, and generalization performance.