FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION OF HIGHER EDUCATION

"NATIONAL RESEARCH UNIVERSITY HIGHER

SCHOOL OF ECONOMICS"

MOSCOW INSTITUTE OF ELECTRONICS AND MATHEMATICS

TECHNICAL SPECIFICATION

EXERCISE

for independent work according to an individual course plan

"Project Workshop "Python in Data Science""

Predicting Personal Income Level Based on Socio-Economic Data

DEVELOPER's GUIDE

Student: Khodja Rinas

Phone number: +79890474940

Email: rkhodzha@edu.hse.ru

Supervisor: Polyakov Konstantin Lvovich

Content table

List of figures

1. Install Prerequisites

Python 3.10

To begin, ensure that Python is available on your system:

- For Windows users: Open Command Prompt and enter: **python --version**
- For Linux or macOS users: Use Terminal and type: **python3--version**

If Python is already installed, you should see output resembling: **Python 3.10.6** or higher.

Python Not Detected?

In case your system doesn't recognize the command:

- Visit the official Python website: https://www.python.org/downloads/
- Choose the latest stable release (ideally version 3.10 or above).
- Launch the installer. Important: Be sure to enable the "Add Python to PATH" option before selecting "Install Now".

Anaconda

To verify whether Anaconda is installed on your system:

- Open a command-line interface (e.g., CMD or Terminal) and execute: **conda --version**

If installed correctly, you'll see an output similar to: **conda 24.x.x**

Anaconda Not Found?

If the command isn't recognized, you'll need to install Anaconda:

- Navigate to the official download page: https://www.anaconda.com/download
- Select the appropriate installer for your operating system.
- Launch the installer and proceed with the default installation options (no changes needed during setup).

2. Project

After downloading Work.zip, locate it (usually in your Downloads folder) and extract it.
Make sure the path is something like this: **YourPath\Work\Work\Scripts (for example)**
You'll get a folder with the following structure:

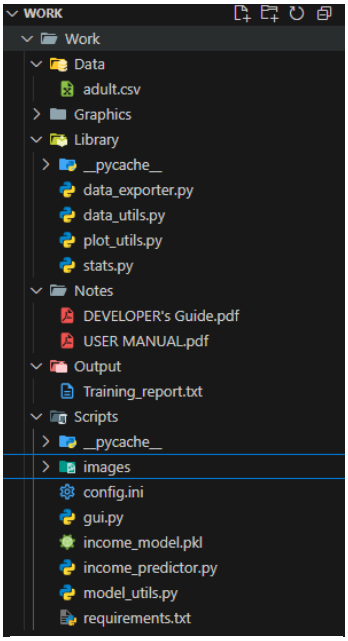

Figure 1 - Project Structure

3. Virtual Environment

After extracting the Zip file, Open **anaconda Prompt**

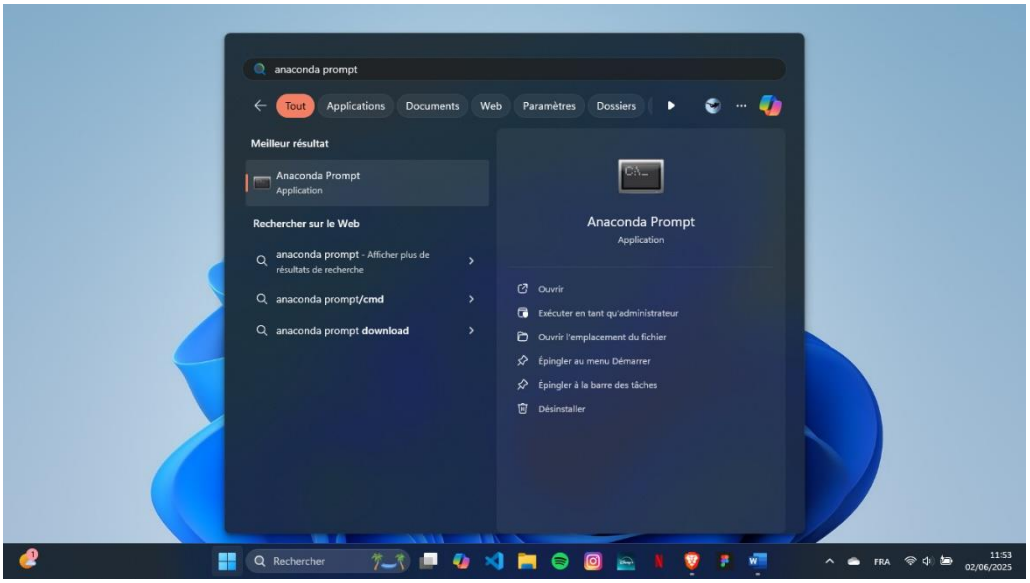

Figure 2 - Anaconda Prompt

Access the path to your extracted Work folder. If the folder was extracted in Downloads, it should be something like: **YourPath\Work\Work**

- Access Folder: **YourPath\Work\Work**
- Create virtual environment: **conda create --name income_predictor python=3.10**
- Activate it: **conda activate income_predictor**
  - **Base** will change into **income_predictor** which means the environment is activated
- Now Acces Folder: **YourPath\Work\Work\Scripts**
  - Install all dependencies: **pip install-r requirements.txt**
- Open Anaconda Navigator and select the environment you created.
- Search for Spyder, install it if needed, then launch it.
- In Spyder, **open income_predictor.py** and **config.ini** from the **Work\Work\Scripts** folder.

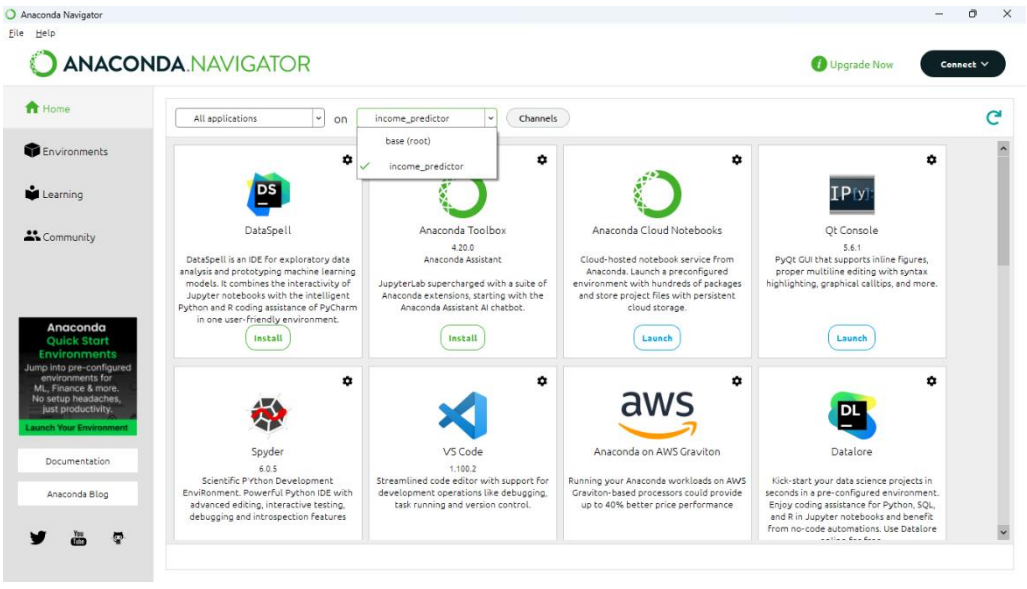

Figure 3 - Environment selection

Within this file, exists the configuration of the app, from here the user can change the colors and fonts of the app.

```
[interface]
window_width = 800
window_height = 1200
bg_color = beige
font_family = Arial
font_size = 12
sidebar_font_size = 14
report_button_color = yellow
button_color = grey
image_max_width = 800
image_max_height' = 500
export_button_color = green
```

Figure 4 - App configuration

4. Python Interpreter

In Spyder, go to **Tools > Preferences > Python Interpreter**, then browse and select the interpreter from the environment you created (e.g., **income_predictor\python.exe**).
Click Apply, and restart Spyder if prompted.

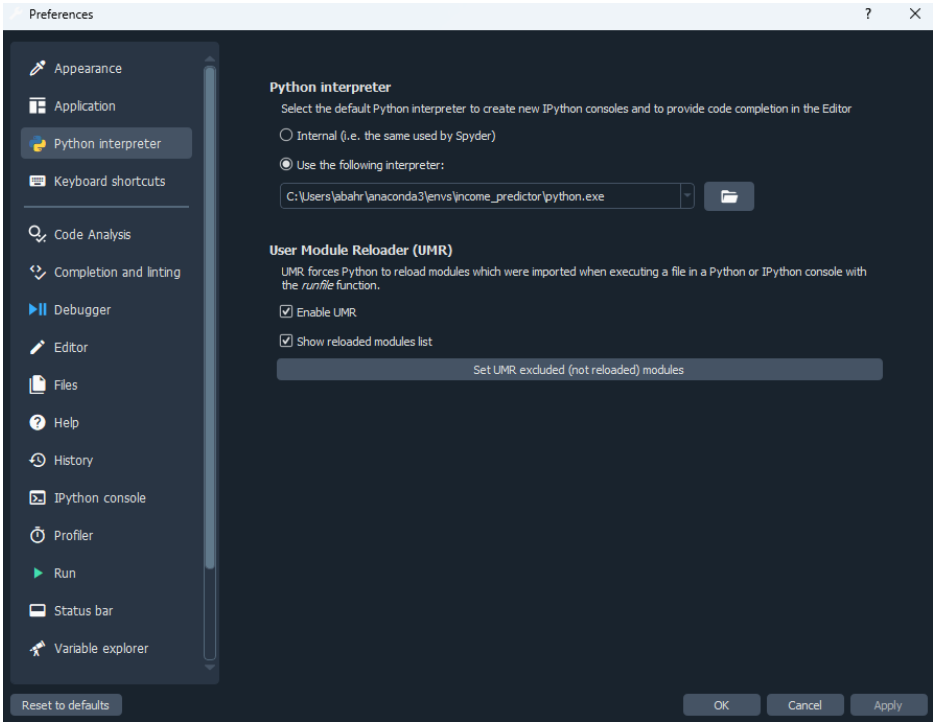

Figure 5 - Python interpreter

You should see the environment we are using at the bottom of Spyder, make sure it's the right one:

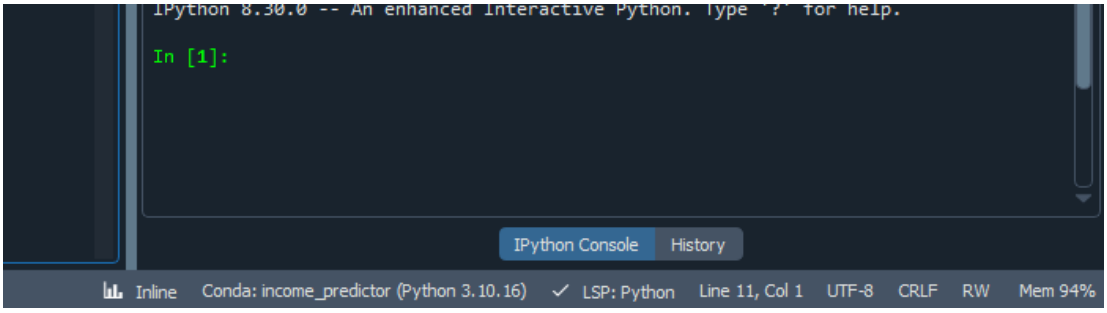

Figure 6 - Environment check

5. Run the Program

Now go to the **income_predictor.py** file opened in Spyder and run it and the GUI must appear

6. Developer's Guide

Work\Work\Library\data_utils.py

load_and_clean_data(csv_path)

Load the raw data from the adult.csv file and clean it to prepare for analysis and modeling. This includes validating column structure and handling formatting issues and placeholder missing values.

1. Define Expected Columns:

   expected_columns = [...]

   - Lists all columns expected in the dataset based on the adult dataset.

2. Load CSV File:

   df = pd.read_csv(csv_path)

   - Uses pandas to load the CSV into a DataFrame.
   - Assumes the file exists and is readable — if not, it raises a FileNotFoundError or ParserError.

3. Validate Column Structure:

   if not all(col in df.columns for col in expected_columns):
   raise ValueError("CSV does not contain the required columns.")

   - Ensures that all expected columns are present in the loaded DataFrame.
   - Raises a ValueError if there's any mismatch, so the issue is caught early (e.g., wrong file passed or modified dataset structure).

4. Clean Object Columns:

   for col in df.select_dtypes(include='object').columns:
   df[col] = df[col].str.strip().replace('?', 'Unknown')

   - Iterates only over string-based (object) columns.
   - For each string column:
     - Strips whitespace (e.g., ' Male ' → 'Male')
     - Replaces '?' (a common missing value placeholder in this dataset) with the literal string 'Unknown'.

5. Return the Cleaned DataFrame:

   return df

inspect_data(df)

To provide a quick and comprehensive overview of the dataset's structure, content, and potential issues. This is particularly useful for debugging, exploratory data analysis (EDA), or verifying that the data was loaded correctly.

1. Print General Info:

   print("\n 📄 Dataset Info:")
   print(df.info())

   - df.info() summarizes the column names, non-null counts, and data types. It helps identify:
     - Columns with nulls (non-null < total rows)
     - Data type mismatches (e.g., numeric values read as strings)

2. Print Dataset Shape:

   print("\n 📏 Shape:", df.shape)

   - Displays the number of rows and columns in the dataset.

3. Count Missing (NaN) Values:

missing_count = df.isna().sum().sum()

print("\n ❓ Missing (NaN) values:", missing_count)

- Uses .isna().sum().sum() to count total NaN entries.

4. Count Entries with ?:

question_marks = (df == "?").sum().sum()

print(" ❓ Entries with '?':", question_marks)

- Explicitly checks for the presence of '?' string entries.

5. Print Descriptive Statistics:

print("\n 📊 Basic Description:")

print(df.describe())

- Provides summary statistics only for numeric columns

features.split_features_target(df)

To separate the input dataset into two distinct parts:

- **Features (features_df):** All columns used to predict the target.
- **Target (target_series):** A binary label indicating whether an individual earns more than $50K.

1. Split Features (X):

features_df = df.drop("income", axis=1)

- Drops the income column from the dataset.

2. Create Target (y):

target_series = df["income"].apply(lambda x: 1 if str(x).strip() == ">50K" else 0)

- Extracts the income column and maps it to binary values:
  - 1 → income is greater than $50K (i.e., ">50K")
  - 0 → otherwise (i.e., "< =50K")
- .strip() ensures no leading/trailing whitespace issues.
- Converts string labels to numeric classification targets.

3. Return Result:

return features_df, target_series

- Returns:
  - features_df: a DataFrame of predictor variables.
  - target_series: a Series of binary target labels (0 or 1).

Work\Work\Library\plot_utils.py

Every function inside this file uses the same libraries **os**, **matplotlib.pyplot** and **seaborn**, These functions have approximately:

1. Prepare Output Directory:

output_dir = _graphics_output_dir(base_dir)

- Constructs an absolute path to the directory where the plot image will be saved.
- Uses a helper function (_graphics_output_dir) to standardize where all plots are stored.

2. Initialize Plot:

plt.figure(figsize=(8, 6))

- Initializes a new matplotlib figure with a fixed size (8 inches by 6 inches).

3. Generate Histogram:

This is the only step step that differs for each plot generation

4. Final Touches and Save:

plt.title(Plot tilte)

plt.tight_layout()

plt.savefig(os.path.join(output_dir, plot_title.png'))

plt.close()

- Adds a title to the plot.
- Uses tight_layout() to auto-adjust margins and avoid cutoff labels.
- Saves the figure as plot_title.png in the output directory.
- plt.close() frees up memory by closing the plot object.

5. Developer Feedback:

print(" ✅ plot_title.png saved")

plot_age_distribution(df, base_dir)

sns.histplot(df['age'], bins=30, kde=False)

- Uses Seaborn to plot a histogram of the age column.
- bins=30: the age range is divided into 30 intervals for granularity.
- kde=False: disables the kernel density estimate, keeping only the raw histogram bars.

plot_hours_per_week_distribution(df, base_dir)

sns.histplot(df['hours-per-week'], bins=30, kde=False)

- Uses Seaborn to plot a histogram of the 'hours-per-week' column.
- bins=30: splits the range of working hours into 30 intervals for better visibility.
- kde=False: disables the smoothed density curve, showing only raw frequency bars.

plot_education_level_pie(df, base_dir)

plt.pie(edu_counts, labels=edu_counts.index, autopct='%1.1f%%', startangle=140)

- Creates a pie chart displaying the percentage distribution of education levels.
- Each slice represents a category's share of the total dataset.
- Provides a compact view of categorical proportions.

plot_workclass_distribution (df, base_dir)

sns.countplot(y='workclass', data=df, order=df['workclass'].value_counts().index)

- Plots the frequency of each 'workclass' category as a horizontal bar chart.
- Categories are ordered by count for clearer comparison.
- Useful to understand the distribution of employment types.

plot_income_distribution (df, base_dir)

sns.countplot(x='income', data=df)

- Plots a simple bar chart showing the count of each income class.
- x-axis represents the income categories ('<=50K', '>50K').
- Helps visualize class imbalance in the dataset.

plot_age_vs_income (df, base_dir)

sns.boxplot(x='income', y='age', data=df)

- Generates a box plot comparing age distributions across income classes.
- Useful for spotting trends, medians, and outliers in age per income group.
- Each box summarizes quartiles and variability of age within each income class.

plot_marital_status_distribution (df, base_dir)

sns.countplot(y='marital-status', data=df, order=df['marital-status'].value_counts().index)

• Creates a horizontal bar chart of the 'marital-status' categories.

• Orders categories by frequency to enhance readability.

• Helps identify dominant marital status groups in the data.

plot_occupation_treemap (df, base_dir)

squarify.plot(sizes=occ_counts.values, label=occ_counts.index, alpha=0.8)

• Creates a treemap where each occupation is represented as a rectangle.

• The size of each box corresponds to the number of people in that occupation.

• Useful for visualizing part-to-whole relationships across many categories.

plot_race_pie (df, base_dir)

plt.pie(race_counts, labels=race_counts.index, autopct='%1.1f%%')

• Generates a pie chart showing the distribution of racial groups.

• Helps visualize demographic breakdown in percentage terms.

• Ideal for comparing the size of discrete categories.

plot_gender_distribution (df, base_dir)

sns.countplot(x='gender', data=df)

• Draws a bar chart showing the count of each gender.

• x-axis maps to gender categories ('Male', 'Female').

• Good for identifying gender imbalance.

plot_education_vs_income (df, base_dir)

sns.countplot(y='education', hue='income', data=df, order=df['education'].value_counts().index)

• Creates a grouped horizontal bar chart by education level and income class.

• hue='income' shows both income groups for each education level.

• Helps analyze income disparities across education backgrounds.

plot_occupation_vs_income_heatmap (df, base_dir)

sns.heatmap(crosstab, annot=True, fmt="d", cmap="YlGnBu")

• Produces a heatmap of occupation vs income group using a cross-tabulation.

• Each cell shows the count of individuals with a given income in a specific occupation.

• Makes it easy to identify income trends by profession.

plot_correlation_heatmap (df, base_dir)

sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f")

• Plots a correlation heatmap of selected numeric features.

• `annot=True`: displays correlation coefficients in each cell.

• `cmap='coolwarm'`: uses a diverging colormap to highlight negative vs positive correlations.

_graphics_output_dir(base_dir)

This function ensures that a valid directory always exists for saving visual output files generated by the plotting functions.

- Returns the absolute path to the graphics output folder.
- Appends 'Graphics' to the given base directory.
- Used internally by all plotting functions to save the plots inside Work\Work\Graphics

Work\Work\Scripts\model_utils.py

build_model_pipeline(numerical_cols, categorical_cols)

1. Numerical Pipeline

   numerical_pipeline = Pipeline(steps=[
   ("imputer", SimpleImputer(strategy="mean")),
   ("scaler", StandardScaler())])

   - Fills missing numerical values using the mean of each column.
   - Standardizes features using Z-score normalization (StandardScaler).

2. Categorical Pipeline

   categorical_pipeline = Pipeline(steps=[
   ("imputer", SimpleImputer(strategy="constant", fill_value="missing")),
   ("onehot", OneHotEncoder(handle_unknown="ignore"))])

   - Replaces missing categorical values with "missing".
   - Encodes categorical features into one-hot vectors, ignoring unknown categories at inference time.

3. Column Transformer

   preprocessor = ColumnTransformer(transformers=[
   ("num", numerical_pipeline, numerical_cols),
   ("cat", categorical_pipeline, categorical_cols)])

   - Applies the right pipeline to each column group (numerical_cols vs categorical_cols).

4. Final Model Pipeline

   model = Pipeline(steps=[
   ("preprocessor", preprocessor),
   ("classifier", RandomForestClassifier(n_estimators=100, random_state=42))])

   - Combines preprocessing with a RandomForestClassifier (100 trees, fixed random seed for reproducibility).

5. Return

   - Returns the complete Pipeline object, ready for .fit() and .predict() calls.

train_model(model, X_train, y_train)

The **train_model** function takes a prepared pipeline and training data as input and fits the model accordingly. It encapsulates the training process, allowing easy reuse and integration into workflows where the model is trained only once and then evaluated or deployed.

evaluate_model(model, X_test, y_test)

The **evaluate_model** function assesses the model's performance using accuracy and a full classification report. It predicts outcomes on the test set and computes standard metrics using **sklearn**, displaying the

results in a readable format. This function provides both printed output and returned values for further use.

save_model(model, path="income_model.pkl")

The **save_model** function uses **joblib** to serialize and store the trained pipeline to disk, making it easy to reuse the model without retraining. By default, it saves the model as **income_model.pkl** and prints confirmation when done.

load_model(path="income_model.pkl")

The **load_model** function complements the previous one by loading a previously saved model from file. It allows immediate restoration of a trained pipeline, ready for predictions or further evaluation.

save_report(df, filename, base_dir, title=None, description=None)

1. Create Output Directory

   output_dir = os.path.join(base_dir, 'Output')

   os.makedirs(output_dir, exist_ok=True)

   - Ensures the target directory Output/ exists inside base_dir.

2. Generate File Path

   file_path = os.path.join(output_dir, filename)

   - Combines base path and filename to form the full destination path.

3. Create Report Header

   now = datetime.now().strftime("%Y-%m-%d %H:%M")

   header = f"{title}\nGenerated on: {now}\n"

   if description:

   header += f"\n{description}\n"

   - Captures current time and adds optional title and description for context.

4. Format DataFrame as Table

   table = tabulate(df, headers='keys', tablefmt='github', showindex=False)

   - Converts the DataFrame to a readable GitHub-flavored markdown table using tabulate.

5. Write to File

   with open(file_path, "w", encoding="utf-8") as f:

   f.write(header + "\n" + table)

   - Writes the full report content (header + table) to the .txt file in UTF-8 encoding.

6. Log Confirmation

   print(f"✅ Report saved to {file_path}")

Work\Work\Scripts\gui.py

- **io, os, sys:** used for file operations, path manipulations, and stdout redirection during inspection.
- **tkinter & submodules:** used for building the main GUI interface.
- **configparser:** handles reading from the config.ini for theme/customization.
- **pandas:** used for loading and manipulating tabular data.
- **PIL.Image, ImageTk:** for displaying plots as images in the GUI.
- **train_test_split:** splits data into training and test sets for model evaluation.
- **sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__)))):** Adds the root directory of the project to the system path to allow module imports from sibling folders (Library, Scripts, etc.).

- Imports data-related utility functions: loading/cleaning CSVs, inspecting data, splitting features and target.
- Imports modeling utilities: pipeline construction, model training, evaluation, saving model & reports.
- Imports plotting functions to visualize various statistical relationships and distributions in the data.
- **CONFIG_PATH = "config.ini":** Path to the config file used to control GUI appearance and behavior.
- **BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__))):** Calculates the root directory of the project — used for relative path management throughout the app.
- **CSV_PATH = os.path.join(BASE_DIR, "DATA", "adult.csv"):** Defines the path to the main CSV data file to be used throughout the GUI.
- Declares UI theme-related global variables. Their values will be populated by **apply_config()** later using **CONFIG_PATH**

load_config()

This function reads the config.ini file and extracts the [interface] section containing UI-related settings (fonts, colors, sizes, etc.). If the section does not exist, it is created by default. The function returns a **ConfigParser** object that allows other parts of the code to access configuration values dynamically.

- Initializes a ConfigParser instance to read INI-format files : config = configparser.ConfigParser()
- Loads the configuration from the file path specified in CONFIG_PATH (usually config.ini).
- Ensures the 'interface' section exists in the config (prevents crashes if missing):

```
if 'interface' not in config:
config['interface'] = {}
```

- Returns the entire config object to be used by other functions like apply_config.

apply_config()

This function applies visual settings retrieved by **load_config** to global variables that control the look and feel of the application. It sets fonts, sizes, and color themes, enabling centralized control of the UI appearance through the config file without editing the source code directly.

- Declares that this function will modify the global variables used across the GUI for style.
- Loads the INI configuration and extracts the 'interface' section.
- Defines a helper function that gets a config value or returns the default if it's empty or missing.
- Applies values from the INI config to global UI parameters with safe fallbacks in case values are missing.

_graphics_output_dir()

This function returns the absolute path to the Graphics directory where all plot images will be saved. If the folder doesn't exist, it is created automatically. This ensures all output images are saved consistently in one place.

- Navigates three levels up from the current file's directory to get the project root, then appends Work/Graphics.

data_page(page_frame)

The **data_page** function builds the layout and behavior of the “Data” tab. It includes interactive buttons, a status bar, and a content frame. This page enables the user to load, inspect, split, and process data, train a model, and export a training report. Each button triggers an internal function that handles a specific operation.

data_page_frame = tk.Frame(page_frame, bg=BG_COLOR)
data_page_frame.pack(fill=tk.BOTH, expand=True, padx=40, pady=30)

- First Creates the main frame for the "Data" tab.
- Fills the entire parent (page_frame) with padding.
- Uses BG_COLOR as the background theme.

data_page_frame.columnconfigure(0, weight=1)
data_page_frame.rowconfigure(1, weight=0)
data_page_frame.rowconfigure(2, weight=1)

- Makes column 0 expandable.
- Row 1 (status bar) stays fixed.
- Row 2 (table area) expands to take remaining space.

button_frame = tk.Frame(data_page_frame, bg=BG_COLOR)
button_frame.grid(row=0, column=0, pady=10, sticky="ew")
button_frame.columnconfigure((0, 1, 2, 3, 4), weight=1)

- A horizontal container placed in the first row (top of the page).
- Evenly distributes space across 5 button slots.

status_label = tk.Label(data_page_frame, text="", bg=BG_COLOR, fg="green",font=(FONT_FAMILY, FONT_SIZE, "bold"))
status_label.grid(row=1, column=0, sticky="ew", pady=5)

- Creates a label to show dynamic messages (e.g. "✅ Data loaded").
- Positioned below the button row.
- Styled with font and colors from config.

content_frame = tk.Frame(data_page_frame, bg=BG_COLOR)
content_frame.grid(row=2, column=0, sticky="nsew")
content_frame.columnconfigure(0, weight=1)
content_frame.rowconfigure(0, weight=1)

- Creates the main content area for showing tables or text widgets.
- Occupies the bottom section of the frame.
- Configured to expand in both directions (nsew)

clear_content(): Clears the content display area so that new widgets (tables, text) can be shown without overlap. It's used before rendering new data.
update_status(text, color="green"): Updates the on-screen status label with a message and color (e.g., green for success, red for error). This gives feedback to the user after an action.
display_table(df): Displays the cleaned dataset as a scrollable table using **Treeview**. Columns are dynamically created based on the **DataFrame**, and data is filled row by row.

handle_load_data(): Loads and cleans the dataset using **load_and_clean_data,** then displays it in the table. It also handles errors and updates the status accordingly.
handle_inspect_data(): Runs **inspect_data** to show a summary of the dataset (shape, missing values, stats). Output is captured and displayed in a scrollable, read-only Text widget.
handle_split_data(): Splits the dataset into features X and target y using split_features_target, then displays their shapes in a summary text box. This is necessary before training.
handle_train_model(): Splits data into train/test sets, builds and trains a model pipeline, evaluates its performance, and displays the classification report. The model is saved automatically.
handle_export_report(): Exports a classification report to a .txt file using **save_report**, including optional title and description. The report is saved in the Output directory.

At the end creates buttons widgets each one inside its button_frame that, when clicked, triggers the function related to it, each button does a different job.

visualization_page(page_frame)

The **visualization_page** function sets up the interface for generating plots. It includes a data loading button, a dropdown menu for selecting plot types, and logic to display the resulting image directly in the window using Pillow.

visualization_page_frame = tk.Frame(page_frame, bg=BG_COLOR)

visualization_page_frame.pack(fill=tk.BOTH, expand=True, padx=40, pady=30)

- Creates the main frame for the "Visualization" tab.
- Fills the entire parent page_frame with padding.
- Uses the theme color BG_COLOR as background.

visualization_page_frame.columnconfigure(0, weight=1)

visualization_page_frame.rowconfigure(1, weight=1)

- Makes column 0 expandable.
- Row 1 (reserved for the status label or image) expands to fill vertical space.

visualization_frame = tk.Frame(visualization_page_frame, bg=BG_COLOR)

visualization_frame.grid(row=0, column=0, pady=10, padx=10, sticky="ew")

- Adds a top sub-frame inside the visualization tab.
- Positioned in row 0, used to contain buttons and dropdown for plot controls.
- Includes padding and expands horizontally (ew).

visualization_frame.columnconfigure((0, 1, 2), weight=1)

- Makes all 3 columns (for buttons and dropdown) share equal horizontal space.

status_label = tk.Label(data_page_frame, text="", bg=BG_COLOR, fg="green",font=(FONT_FAMILY, FONT_SIZE, "bold"))
status_label.grid(row=1, column=0, sticky="ew", pady=5)

- Creates a label to display plot status messages (e.g. ✅ Plot loaded).
- Styled using config settings.
- Placed in row 1, left-aligned horizontally (w).

update_status(text, color="green"): Updates the on-screen status label with a message and color (e.g., green for success, red for error). This gives feedback to the user after an action.
handle_load_data(): Loads and cleans the dataset using **load_and_clean_data,** then displays it in the table. It also handles errors and updates the status accordingly.
handle_plot_selection(): Handles plot generation based on the selected plot name. It calls the appropriate function (e.g., **plot_age_distribution**), generates the plot image, and displays it in the window. If the image can't be found or an error occurs, a message box alerts the user.

configuration_page(page_frame)

The **configuration_page** function renders the interface customization tab. It allows users to adjust visual preferences such as font styles, button colors, and background theme. Settings are saved and applied upon restart of the application.

configuration_page_frame = tk.Frame(page_frame, bg=BG_COLOR)

configuration_page_frame.pack(fill=tk.BOTH, expand=True, padx=40, pady=30)

- Creates the main frame for the "Configuration" tab.
- Fills the parent **page_frame** with internal padding.
- Uses the background color defined in **BG_COLOR**.

configuration_page_frame.columnconfigure(0, weight=1, minsize=150)

configuration_page_frame.columnconfigure(1, weight=2, minsize=200)

- Column 0 holds the labels; column 1 holds the input widgets.
- Column weights and minimum sizes ensure a balanced layout.

add_field(label_text, key, options=None, default="", is_dropdown=True)

- Helper function to add a labeled dropdown or entry field to the page.
- If options are provided, a ttk.Combobox is used; otherwise a text entry.
- Fields are automatically stacked vertically by row.
- Special logic is used to convert the selected background name into a hex code.

on_save()

- Gathers all selected configuration values and updates the config file.
- Maps selected background name to its corresponding hex code.
- Triggers a full application restart using os.execl to apply changes.
- Displays a messagebox to notify the user that a restart is required.

save_btn = tk.Button(...)

save_btn.grid(row=row, column=0, columnspan=2, pady=(25, 0))

- Adds a final "Save Configuration" button at the bottom.
- Spans two columns to align with the field layout.
- Styled according to user-defined theme variables.

launch_gui()

This is the entry point of the GUI application. It initializes the main Tkinter window, applies the UI configuration, and sets up the animated sidebar with navigation icons. The sidebar allows users to switch between the data page and visualization page seamlessly.

apply_config()

- Loads and applies UI styling (fonts, colors) from config.ini.

```
root = tk.Tk()
root.geometry('900x600')
root.title('Tkinter Income Predictor)
```

- Creates the main Tkinter window with fixed size 900x600 and a title.

```
toggle_icon = tk.PhotoImage(file='images/open_menu.png')
close_icon = tk.PhotoImage(file='images/close_menu.png')
data_icon = tk.PhotoImage(file='images/data.png')
visualization_icon = tk.PhotoImage(file='images/visualization.png')
```

- Loads images to be used for sidebar toggle, data tab, and visualization tab icons.

switcher(ind, page, pg): Changes the content shown in the main frame depending on which sidebar button is clicked. It also visually highlights the selected option.

```
data_button_ind.config(bg=SIDEBAR_COLOR)
visualization_button_ind.config(bg=SIDEBAR_COLOR)
ind.config(bg='white')
```

- Resets button indicators to default color, highlights the selected one.

```
if menu_sidebar_frame.winfo_width() > 50:
    shrink_menu_sidebar()
```

- Auto-shrinks the sidebar if it's expanded when a button is clicked

```
for frame in page_frame.winfo_children():
    frame.destroy()
page(pg)
```

- Clears the current page content and loads the new one by calling page(pg).

expand_menu(): Grows the sidebar width gradually to 250 pixels using after() animation loop.

expand_menu_sidebar(): Triggers expand_menu() and switches the toggle icon to "close".

shrink_menu(): Shrinks the sidebar back down to 50 pixels with smooth animation.

shrink_menu_sidebar(): Triggers shrink_menu() and swaps toggle icon back to "open".

```
page_frame = tk.Frame(root, bg=BG_COLOR)
page_frame.place(relwidth=1.0, relheight=1.0, x=30)
data_page(page_frame)
```

- Creates the main content area (right side of the UI).

- Takes up nearly the entire window (relwidth=1.0), shifted by 30 pixels to make room for sidebar.
- Loads the data tab by default.

menu_sidebar_frame = tk.Frame(root, bg=SIDEBAR_COLOR)

- Defines the left sidebar frame (navigation panel), styled with theme color.

toggle_button = tk.Button(...)

- The small hamburger menu button at the top-left.
- Toggles sidebar between expanded/collapsed states.

data_button = tk.Button(...)
data_button_ind = tk.Label(...)
data_button_title = tk.Label(...)

- Button with data icon, triggers data_page.
- data_button_ind: vertical highlight strip to indicate selection.
- data_button_title: text label ("Data manipulation") appears when sidebar expands.
- Clicking the icon or label switches the page.

visualization_button = tk.Button(...)
visualization_button_ind = tk.Label(...)
visualization_button_title = tk.Label(...)

- Same structure as above, but for "Visualizations" page.
- Calls visualization_page when clicked.

menu_sidebar_frame.pack(side=tk.LEFT, fill=tk.Y, padx=3)
menu_sidebar_frame.pack_propagate(flag=False)
menu_sidebar_frame.configure(width=50)

- Packs the sidebar on the left edge of the window.
- Prevents auto-resizing (pack_propagate(False)).
- Sets initial collapsed width to 50 pixels.

```
try:
    root.mainloop()
except Exception:
    pass
```

- Starts the Tkinter event loop.