Assignment (Term Paper)

Statistical Programming with Python, Summer Term 2026

May 22, 2026

Please read the provided `instruction.pdf` file carefully!

- The term paper is to be submitted alone or by teams of up to **4** students.
- You can export both Quarto markdown and Jupyter notebooks to PDF.
- The submission deadline is **June 8, 2026, 23:59 (German time)**.
- Each group is required to submit an independent solution; collaboration between groups is not allowed.
- The title page must list the names and student IDs of **all** group members.
- The paper should include *replicable*, *self-explanatory* code and readable documentation: explain statistical questions and code logic, and summarize results. Use inline comments as needed.
- You may answer in German or English. Choose your software (Jupyter, Quarto, Word, LaTeX); Quarto or Jupyter is recommended.
- If you do not have a group, you can email the instructor.
- Attempt as many exercises as possible, even if your solutions are incomplete.

Grading will consider:

- Code correctness and quality
- Layout and organization
- Presentation quality (nice tables/figures)
- Conciseness and clarity of explanations

Section 1: Classification on Survey Data (50 Points)

In this section you will work with an extract from the **Politbarometer** (ZA8974), a long-running German political survey conducted by GESIS. The dataset `politbarometer_data.csv` (in the `data_assignment/` folder) contains approximately 26,000 respondents. Your goal is to clean and enrich the data, then build classifiers that predict which party a respondent voted for in the 2021 Bundestag election.

The **target variable** is `vote_class`, which takes the values `CDU_CSU`, `SPD`, `GRUENE`, `FDP`, `AFD`, `THE_LEFT`, and `OTHER`. A full variable description is provided in `variable_overview.csv`.

The following table summarises the key variables:

Variable	Survey question (English)	Type
`state`	Federal state eligible to vote in	categorical
`region_type`	Former West or East Germany	categorical
`municipality_size`	Approximate inhabitants (e.g. `"up to 5,000 inhabitants"`)	categorical string
`sex`	Gender	categorical
`age_group`	Age in bands (e.g. `"30 to 34 years"`)	categorical string
`school_degree`	Highest school-leaving certificate	categorical
`vocational_training`	Completed vocational training	categorical
`employment_status`	Current employment status	categorical
`occupational_status`	Occupational position	categorical
`household_size`	Number of people in household	numeric
`household_adults`	Number of adults in household	numeric
`household_union_members`	Union membership in household	categorical
`religion`	Religious affiliation	categorical
`party_id_exists`	Party identification (Yes / No)	categorical
`party_id_party`	Party identified with	categorical
`party_id_strength`	Strength of party identification	categorical
`interview_week`	Survey week identifier	numeric

Variable	Survey question (English)	Type
`vote_class`	Recall vote in 2021 Bundestag election	**target**

Exercise 1: Data Wrangling and Exploration (25 Points)

1. Load the dataset. Report the number of observations and variables. Print the class distribution of `vote_class` (counts **and** percentages). For each column, compute the number and percentage of missing values and display the result as a sorted table. *(3 pts)*

2. Several columns have substantial missing rates. For each column with at least one missing value, decide on one of three strategies: **drop the column**, **fill with an informative placeholder** (e.g. `"none"`), or **impute** (median for numeric, mode for categorical). Justify every decision in a table or brief paragraph, then apply all decisions. Verify that only a handful of columns retain missings and list them. *(5 pts)*

   > **Tip**
   >
   > Columns missing more than 35 % of observations are generally too sparse to impute meaningfully — dropping them is safer. For `party_id_party` and `party_id_strength`, ask yourself whether "missing" is itself informative (the respondent has *no* party identification); if so, fill with `"none"` rather than imputing a mode.

3. The `municipality_size` column contains ordered strings such as `"up to 5,000 inhabitants"` or `"over 500,000 inhabitants"`. Parse it into **two** new columns: *(4 pts)*

   a. `municipality_upper_bound` — the upper bound as a `float`. For `"over 500,000 inhabitants"` use `500001.0` so it ranks above all `"up to …"` bands.
   b. `urbanization_class` — a three-level categorical: `"rural_small"` (20,000), `"semi_urban"` (100,000), `"urban"` ($>$ 100,000).

   Write a function to handle the parsing. Report the distribution of `urbanization_class`.

   > **Tip**
   >
   > Use `re.findall(r'\d+', text)` to extract all digit sequences from the string. The word `"over"` at the start signals the open-ended top category.

4. The `age_group` column contains strings such as `"30 to 34 years"` or `"80 years and older"`. Parse it into **three** new columns: *(3 pts)*

a. `age_lower_bound` and `age_upper_bound` (both `float`; leave `age_upper_bound` as `NaN` for `"80 years and older"`).
b. `age_midpoint` — the arithmetic midpoint; use `85.0` for the open-ended top band.
c. `age_group_4` — a coarse four-level binning: `"18_34"`, `"35_49"`, `"50_64"`, `"65_plus"` (cut on `age_lower_bound`).

Print the unique `age_group` → `age_midpoint` / `age_group_4` mapping to verify your parser covers all levels.

5. The `occupational_status` column contains fine-grained job categories. Recode them into five broader groups and store the result in a new column `occupational_group`: *(2 pts)*

New group	Original categories (examples)
`managers`	`"Employees, managers"`
`employees`	`"Employees, upscale"`, `"Employees, easy"`, `"Employees, not specified"`
`public_service`	`"Civil servants, …"`, `"Officials, …"`, `"Judge"`
`skilled_blue_collar`	`"Skilled workers"`, `"workers"`, `"Master"`
`self_employed`	`"Self-employed people"`, `"Farmers, self-employed"`
`other`	everything else

Print `df["occupational_status"].value_counts()` first to see all categories, then build a mapping dictionary. Report the count per `occupational_group`.

6. A supplementary file `bundesland_context_demo.csv` (in `data_assignment/`) provides state-level indicators: *(3 pts)*

Column	Description
`state_merge_key`	Lowercase ASCII slug (e.g. `"bavaria"`, `"berlin_west"`)
`unemployment_rate_pct_2024`	Unemployment rate in % (2024)
`gdp_per_capita_eur_2023`	GDP per capita in EUR (2023)
`population_density_per_km2_2023`	Population density per km² (2023)
`east_west`	`"East"` or `"West"`

Create a `state_merge_key` from the `state` column (lowercase, transliterate German umlauts to ASCII digraphs, replace spaces and hyphens with underscores). Merge the context table into the survey data on this key using a left join. Report the merge

success rate (share of rows with a non-missing `gdp_per_capita_eur_2023`) and list any unmatched state labels.

> **Tip**
>
> Berlin is split into two survey categories: `"Berlin East"` / `"Berlin-Ost"` → key `"berlin"` (East), and `"Berlin West"` / `"Berlin-West"` → key `"berlin_west"` (West). The context file has one row per key, so the mapping must distinguish East from West Berlin before the slug replacement.
> Transliteration: `ä` → `ae`, `ö` → `oe`, `ü` → `ue`, `ß` → `ss`. Then replace all non-alphanumeric characters with `_` and strip leading/trailing underscores.

7. Create at least **three** publication-quality visualisations that characterise the enriched dataset before modelling. Suggested plots: *(5 pts)*

   a. **Age distribution by party** — a violin or box plot of `age_midpoint` by `vote_class`, with parties ordered by their median respondent age.
   b. **Regional wealth vs. age** — a state-level scatter plot with interpolated median respondent age on the x-axis and `gdp_per_capita_eur_2023` on the y-axis; use point size proportional to the number of respondents and colour to distinguish East vs. West Germany. Label each state.
   c. **Occupational composition by state** — please refer to Lecture 2 and be creative.

   Briefly (2–3 sentences) interpret each visualisation.

> **Tip**
>
> For (a), sort the x-axis by computing the (weighted or unweighted) median of `age_midpoint` per party and passing `category_orders` to `px.violin`. For (b), use `px.scatter` with `text="state"` to add state labels.

Exercise 2: Multiclass Classification (25 Points)

Using the enriched dataset from Exercise 1, build and evaluate classifiers that predict `vote_class`.

1. **Conceptual: Bagging and Random Forests** *(4 pts)*

   Explain the following in your own words (one paragraph per sub-question):

   a. What does a single deep decision tree learn, and what is its key weakness?
   b. How does **Bootstrap Aggregation (Bagging)** reduce prediction error? Why does averaging many trees help?

c. Random Forests add one extra randomisation step on top of Bagging. What is this step, and why does it reduce the correlation between trees compared to plain Bagging?

2. **Feature matrix and the age encoding problem** *(5 pts)*

   Define a feature matrix `X` and target vector `y`. Drop all rows with a missing `vote_class`.

   The enriched dataset contains two age representations derived from the same source:

Column	Type	Description
`age_midpoint`	numeric	Midpoint of the reported age band (e.g. 54.5 for "50 to 59 years")
`age_group_4`	categorical	Coarse 4-level binning: `18_34`, `35_49`, `50_64`, `65_plus`

   a. Why might including **both** simultaneously be problematic — discuss *separately* for Random Forest and for Logistic Regression.
   b. Choose **one** representation for your final feature set and justify your choice.
   c. List all columns you include in `X` and report the final sample size.

3. **Logistic Regression regularisation** *(4 pts)*

   Read the `sklearn` documentation for `LogisticRegression` (run `help(LogisticRegression)` or visit the online docs).

   a. Which regularisation strategy should you choose if you expect **many features with small but individually significant effects** on vote choice? Explain why it is appropriate here and how it differs from the alternative.
   b. How does the hyperparameter `C` control regularisation strength? In which direction should you adjust `C` if the model is overfitting?
   c. Why is `class_weight='balanced'` relevant for this dataset? Use the class distribution from Exercise 1, Q1 to support your answer.

4. **Preprocessing pipelines** *(5 pts)*

   Build `sklearn` `Pipeline` objects for both a **Random Forest** and a **Logistic Regression** classifier using `ColumnTransformer` to handle numeric and categorical columns separately.

   - **Random Forest** — tree-based splits are scale-invariant; apply only median imputation to numeric columns and `SimpleImputer(strategy="most_frequent")` + `OneHotEncoder(handle_unknown="ignore")` to categorical columns.

- **Logistic Regression** — the gradient-based solver is sensitive to feature scale; apply median imputation **and** `StandardScaler` to numeric columns, plus the same categorical encoding as above.

Print the pipeline structure for each classifier (e.g. `print(pipeline)`).

5. **Fit, predict, evaluate** *(4 pts)*

   Perform an 80 / 20 stratified train-validation split (`stratify=y`, `random_state=42`). Fit both classifiers on the training set. Report:

   a. **Balanced accuracy** for each model (use `sklearn.metrics.balanced_accuracy_score`).
   b. The full `classification_report` with per-class precision, recall, and F1-score.

> Tip
>
> Balanced accuracy is the mean recall across all classes — it is more informative than plain accuracy when class sizes differ substantially. Use `zero_division=0` in `classification_report` to suppress warnings for classes with no predicted samples.

6. **Interpretation** *(3 pts)*

   Answer the following in 2–3 sentences each:

   a. Which model performs better overall? Is the difference large enough to be practically meaningful?
   b. Which `vote_class` is hardest to predict across both models? Propose at least one substantive reason why.
   c. Name one limitation of the current evaluation setup (e.g. single split vs. cross-validation, potential label leakage from `party_id_party`, representativeness of the validation set, etc.).

Section 2: Regression, Overfitting, and Regularization (30 Points)

In this section you will work with the **German City Districts Dataset** (`german_city_districts_data.csv`, in `data_assignment/`). The dataset covers 95 urban districts across major German cities and contains counts of hospitality venues (restaurants, bars, cafés), socio-economic indicators, and geographic coordinates. The goal is to predict the density of restaurants per 100,000 residents in each district based on the available district characteristics.

The following table describes all variables:

Variable	Description	Type
`city`	City name	string
`district_name`	District name	string
`restaurants`	Number of restaurants	numeric
`bars`	Number of bars	numeric
`pubs`	Number of pubs	numeric
`fast_food`	Number of fast food outlets	numeric
`cafes`	Number of cafés	numeric
`transport_stations`	Number of public transport stations	numeric
`proximity_to_center_km`	Distance to city center (km)	numeric
`total_hospitality`	Total hospitality venues	numeric
`bar_to_restaurant_ratio`	Bars-to-restaurants ratio	numeric
`population`	District population	numeric
`area_sqkm`	District area (km^2)	numeric
`population_density`	Residents per km^2	numeric
`median_age`	Median age of residents	numeric
`income_eur_pp`	Average income per person (EUR)	numeric
`share_young_adults_pct`	Share of 18–35 year olds (%)	numeric
`employment_rate_pct`	Employment rate (%)	numeric
`tourism_intensity_per1k`	Tourism intensity per 1,000 residents	numeric
`landuse_mix_entropy`	Land use diversity (entropy index, 0–1)	numeric
`pct_builtup_area`	Share of built-up area (%)	numeric
`walkability`	Street intersections per km^2	numeric
`dist_nearest_univ_km`	Distance to nearest university (km)	numeric
`nightlife_proxy_index`	Nightlife activity proxy index	numeric
`lat`	Centroid latitude	numeric
`lon`	Centroid longitude	numeric

Exercise 3: Hospitality Density Regression (30 Points)

1. **Data overview and cleaning** — Load the dataset. Report its shape and list all column names. Compute the number of missing values per column. Check for duplicate rows and drop them if present. Print a `describe()` summary for all numeric variables. *(3 pts)*
2. **Outcome construction** *(3 pts)*
   a. Add the values of column `pubs` to column `bars`, then drop `pubs`.
   b. Compute three per-capita outcome variables:

$$\texttt{restaurants\_per\_100k} = \frac{\texttt{restaurants}}{\texttt{population}} \times 100{,}000$$

   and analogously `bars_per_100k` and `cafes_per_100k`. Drop the raw count columns `restaurants`, `bars`, and `cafes` after constructing the rate outcomes.

> **Tip**
>
> Also exclude `fast_food`, `total_hospitality`, `bar_to_restaurant_ratio`, `area_sqkm`, and `population` from your feature matrix later — they either directly encode the outcome or would cause data leakage.

3. **Visualisation** *(5 pts)*
   a. Compute the correlation matrix of all numeric variables and visualise it as a **heatmap** using `seaborn.heatmap` with a diverging colour palette (`cmap="coolwarm"`, `center=0`). List the five variables most strongly (absolutely) correlated with `cafes_per_100k`.
   b. Using the `lat` and `lon` columns, create an **interactive Plotly map** of all 95 districts. Requirements:
      - Place each district as a marker at its centroid (`lat`, `lon`).
      - Colour markers by the selected outcome; add a **dropdown menu** to switch between `restaurants_per_100k`, `bars_per_100k`, and `cafes_per_100k`.
      - The hover tooltip must show: district name, city, `transport_stations`, `employment_rate_pct`, `tourism_intensity_per1k`, `population_density`, and all three per-100k values.

> **Tip**
>
> Use `plotly.graph_objects.Scattermap` with `map_style="carto-positron"` (no API key required). Build one trace per outcome (only the first trace visible), then use `updatemenus` buttons with `method="update"` to toggle visibility.
> One district has incorrect coordinates — identify it by inspecting the map. Correct the coordinates manually and re-run the plotting.

4. **Baseline linear regression** *(4 pts)*

   Define a feature matrix `X` using the remaining numeric predictors (excluding `lat`, `lon`, `city`, `district_name`, and any outcome columns). Perform an 80 / 20 train-test split (`random_state=42`). Fit `sklearn.linear_model.LinearRegression` for `cafes_per_100k` and report train/test RMSE and R^2.

   Additionally, fit the same model with `statsmodels.OLS` on the training data (use standard (non-robust) standard errors via `.fit(cov_type="nonrobust")`). Print the 10 predictors with the lowest p-values. Discuss (2–4 sentences) whether the OLS results align with the correlation analysis from (3.a) and whether the significant (alpha = 5%) coefficients make sense.

5. **Polynomial model** *(4 pts)*

   Expand the feature matrix with `sklearn.preprocessing.PolynomialFeatures(degree=3, include_bias=False)`. Fit OLS on the expanded features and report:

   a. The number of features before and after expansion.
   b. Train/test RMSE and R^2.

   Explain (3–5 sentences) why polynomial OLS fails here. What happens mathematically when $p > n$ (more features than observations)?

6. **Regularized alternatives** *(3 pts)*

   Fit **Ridge**, **Lasso**, and **Elastic Net** using the degree-3 polynomial features. Each pipeline must include `StandardScaler` between the polynomial expansion and the regularized estimator. Use the same train-test split as in (4.). Compile a comparison table with columns `model`, `rmse_train`, `rmse_test`, `r2_train`, `r2_test`, sorted by ascending `rmse_test`. Include the baseline OLS and polynomial OLS rows for reference.

> **Tip**
>
> Start with `Ridge(alpha=2.0)`, `Lasso(alpha=0.5, max_iter=50000)`, and `ElasticNet(alpha=0.5, l1_ratio=0.5, max_iter=50000)`. Always scale features before regularization — the penalty is applied to the raw coefficient magni-

tudes, so unscaled features lead to arbitrarily large or small penalties per predictor.

7. **Manual cross-validation for Lasso** *(3 pts)*

   Implement a 10-fold cross-validation loop (using `sklearn.model_selection.KFold(n_splits=10, shuffle=True, random_state=42)`) to select the best `alpha` for Lasso from a grid of 40 values log-spaced between 10^{-4} and 10^{1}.

   a. For each `alpha`, record the mean and standard deviation of the 10 fold-level MSE values.
   b. Plot mean CV MSE (y-axis) against `alpha` (x-axis, log scale) and mark the best `alpha` with a vertical dashed line.
   c. Refit Lasso with the best `alpha` on the full training set. Report its test RMSE and compare it to the fixed-`alpha` Lasso from (6.).

> **Tip**
>
> Wrap the full preprocessing pipeline (imputer → polynomial → scaler → Lasso) inside the CV loop so that each fold's validation set is never used to fit the scaler or polynomial terms. Use `Pipeline` to keep this clean.

8. **Conceptual: Ridge vs. Lasso and coefficient comparison** *(5 pts)*

   a. Explain in 100–150 words the geometric reason why Lasso can set some coefficients *exactly* to zero while Ridge cannot. You may reference the penalty-geometry diagram discussed in the lecture (Lecture 3 Slide 82).

   b. Using the best-`alpha` Lasso from (7.), refit on `bars_per_100k` and `restaurants_per_100k` (use the same `alpha`). Extract the coefficients for all three outcomes and create a side-by-side dot plot — one subplot per outcome — showing the top 25 predictors ranked by mean absolute coefficient across outcomes (i.e., for each predictor, average of the three absolute coefficients: restaurants, bars, cafés). Add a vertical line at zero. Discuss (2–4 sentences): which predictors drive all three outcomes similarly, and where do they diverge?

> **Tip**
>
> After fitting, call
>
> ```
> pipeline.named_steps["poly"].get_feature_names_out(feature_cols)
> ```
>
> to recover human-readable polynomial feature names (e.g. `"tourism_intensity_per1k^2"`, `"median_age nightlife_proxy_index"`). Use `plotly.subplots.make_subplots` with `shared_yaxes=True` for the coefficient comparison plot.