Statistical Methods in AI - Monsoon 2025

Assignment 1

Deadline : 26 August 2025 11:59 P.M.

Instructors: Prof Ravi Kiran Sarvadevabhatla, Prof Saikiran Bulusu

General Instructions

- Your assignment must be implemented in Python.
- Clearly label and organize your code, including comments that explain the purpose of each section and key steps in your implementation. Ensure that your files are well-structured, with headings, subheadings, and explanations as necessary.
- Make sure to test your code thoroughly before submission to avoid any runtime errors or unexpected behavior.
- Your assignment will be evaluated not only based on correctness but also on the quality of code, the clarity of explanations, and the extent to which you've understood and applied the concepts covered in the course.
- We are aware of the possibility of submitting the assignment late in GitHub Classroom using various hacks. Note that we will have measures in place accordingly and anyone caught attempting to do the same will be given a straight zero in the assignment.

AI Tools Usage Instructions (Mandatory if Applicable)

We are aware how easy it is to write code and solve questions with the help of LLM services, but we strongly encourage you to figure out the answers yourself. If you use any AI tools such as ChatGPT, Gemini, Claude, etc., to assist in solving any part of this assignment:

- **If you are unable to explain any part of the solution/code during evaluations, that solution/code will be considered plagiarized and you will be penalized.** You must also be able to briefly explain how you modified or verified the AI-generated content.
- You must include a **shareable link** to the AI Tool chat history or a **screenshot** of the relevant conversation. **Do NOT share any private or sensitive personal information** in the AI Tool conversations you include.

Submission Instructions

- Submit a self-contained **Jupyter notebook** containing all code, plots, and printed tables.
- Report all the analysis, comparison and any metrics in the notebook or a separate report that is part of the submission itself. No external links to cloud storage files, wandb logs or any other alternate will be accepted as part of your submission. Only the values and visualizations as part of your commits will be graded.
- Use your institute email ID to generate a **personalized random seed**:
  - Get the part before `@` in your IIITH email e.g., `username` from `username@program.iiit.ac.in`.
  - Use the SHA-256 hash of this string to ensure uniqueness:

    ```
    import hashlib
    seed = int(hashlib.sha256(username.encode()).hexdigest(), 16)
       % (2**32)
    ```

  - Use this `seed` in all random number generators (e.g., `np.random.default_rng(seed)`).
- All plots must include your **email username** in the title or filename

  ```
  plt.text(
      0.95, 0.95, "username",
      ha='right', va='top',
      transform=plt.gca().transAxes,
      fontsize=10, color='gray', alpha=0.7
  )
  ```

Submission Policy (GitHub Classroom Assignments)

To encourage consistent progress and discourage last-minute submissions, the following policy applies:

- **Minimum Progress Requirement:** You must push at least **two meaningful commits on different days** prior to the deadline.
- **Commit Timing Check:** If over 80% of commits are made within the last 24 hours before the deadline, a **10% penalty** will be applied.
- **Commit Quality:** Commits must reflect actual progress. Non-informative or placeholder commits will not count.
- **Final Submission:** Your final grade is based on the latest commit before the deadline, but commit history will be reviewed.

Guidelines for Implementation and Code Design

1. Use object-oriented programming. For Q1, all sampling and analysis must use the original dataset stored in the `StudentDataset` object.
2. Use appropriate visualization libraries such as `matplotlib`, `seaborn`, or `plotly`. Each plot must include a **title, x-label, y-label, and legend (if applicable)**. Answers with missing labels or legends will receive **0 marks**.
3. Keep visualization and computation logic in **separate functions**.
4. Add `docstrings` to methods explaining what they do.
5. Use reproducible random sampling using the given `seed`.

Q1.0 Dataset Generation [6 marks]

Generate 10,000 student records with the following attributes:

- **gender:** Male (65%), Female (33%), Other (2%) [1]
- **major:** B.Tech (70%), MS (20%), PhD (10%) [1]
- **program:** distribution conditioned on major: [2]

Major	CSE	ECE	CHD	CND
B.Tech	40%	40%	10%	10%
MS	30%	30%	20%	20%
PhD	25%	25%	25%	25%

- **GPA:** Normally distributed by major, clipped to [4.0, 10.0] [2]

Major	GPA Distribution ($\mathcal{N}(\mu, \sigma)$)
B.Tech	$\mathcal{N}(7.0, 1.0)$
MS	$\mathcal{N}(8.0, 0.7)$
PhD	$\mathcal{N}(8.3, 0.5)$

You must implement a class as follows.

```
class StudentDataset:
    def __init__(self, num_students: int, seed: int):
    # Generates the full dataset during initialization using the
        specified number of students and seed.
    def get_full_dataframe(self) -> pd.DataFrame:
    # Do not regenerate the dataset in different methods or cells.
    # Use this method to access the full dataset consistently.
    def generate_gender(self) -> list[str]: ...
    def generate_major(self) -> list[str]: ...
    def generate_program(self, majors: list[str]) -> list[str]: ...
    def generate_gpa(self, majors: list[str]) -> list[float]: ...
    def assemble_dataframe(self) -> pd.DataFrame: ... # Assemble the
        full dataset from gender, major, program, and GPA.
```

Q1.1 Dataset Analysis

(a) Visualizations [15 marks]

Create suitable visualizations for the following distributions. The visualizations should convey meaningful information about the data.

- gender [1]
- major [1]
- program [1]
- GPA [1]
- program conditioned on major [1]
- GPA conditioned on major [1]
- GPA conditioned on program [1]
- GPA conditioned on program and major [2]
- gender, major, program and GPA of 100 randomly sampled students [3]
- Summary of entire dataset(e.g. pairplots) [3]

You may implement the following methods:

```
def plot_gender_distribution(self) -> None: ...
def plot_major_distribution(self) -> None: ...
def plot_program_distribution(self) -> None: ...
def plot_gpa_distribution(self, bins: int = 20) -> None: ...
def plot_program_by_major(self) -> None: ...
def plot_gpa_by_major(self) -> None: ...
def plot_gpa_by_program(self) -> None: ...
def plot_gpa_by_program_and_major(self) -> None: ...
def plot_sampled_dataset(self) -> None: ...
def plot_entire_dataset_summary(self) -> None: ...
```

(b) GPA Summary Statistics [1 mark]

Define a method to compute the mean and standard deviation of GPA:

```
def gpa_mean_std(self) -> tuple[float, float]: ...
```

Report the results and briefly comment on any observations.

(c) Program-Major Combinations [2 marks]

Define a method to count the number of students for each unique (program, major) pairs. Also write a method to visualize it with a heatmap.

```
def count_students_per_program_major_pair(self) -> pd.Dataframe: ...
def visualize_students_per_program_major_pair(self, counts_df : pd.
    Dataframe) -> None: ...
```

Report the counts and describe any patterns you observe.

Q1.2 Simple vs Stratified Sampling [5 marks]

- Sample 500 students uniformly at random. Repeat 50 times. Calculate the mean GPA and standard deviation for each sample. Report the average mean GPA and the standard deviation of the mean GPAs across the 50 repetitions. [2]
- Repeat using stratified sampling by major (divide population into strata by major based on their proportion in the dataset). Compare the results. [2]
- Which method has lower std deviation? Why? [1]

```
def get_gpa_mean_std_random(self, n: int = 500, repeats: int = 50)
    -> tuple[float, float]: ...
def get_gpa_mean_std_stratified(self, n: int = 500, repeats: int =
    50) -> tuple[float, float]: ...
```

Q1.3 Gender-Balanced Cohort [5 marks]

- Sample 300 students with exact same count across genders. Repeat 5 times. Report gender counts. [1]
- Consider the following **Sampling Strategy A**: Randomly pick a value from a discrete set of categories with equal probability (here, gender). Randomly pick a student from that category. Sample 300 students using this sampling strategy. Repeat 5 times. Report gender counts. [2]

- For each sample size (300, 600, 900, 1200, 1500), repeat the above sampling process 10 times. In each repeat, count the number of students in each gender, find the difference between the largest and smallest counts, and divide by the sample size (n) to get the maximum relative difference. Compute the average of these values over the 10 repeats for each sample size. Plot a histogram of average maximum relative difference (y-axis) versus sample size (x-axis). [2]

```
def get_gender_balanced_counts(self, n: int = 300, repeats: int = 5)
    -> list[dict[str, int]]: ...
def sample_gender_uniform_random(self, n: int = 300, repeats: int =
   5) -> list[dict[str, int]]:
def plot_avg_max_gender_diff_vs_sample_size(self, sample_sizes: list
   [int], repeats: int = 10) -> None: ...
```

Q1.4 GPA-Uniform Cohort [3 marks]

- Using Sampling Strategy A, select 100 students such that their GPA values are approximately uniformly distributed across 10 bins. [1]
- Plot GPA histogram and compare to original dataset's histogram. [1]
- Did you sample with or without replacement? Why? [1]

```
def sample_gpa_uniform(self, n: int = 100, bins: int = 10) -> pd.
   DataFrame: ...
def plot_gpa_histogram_comparison(self, sampled_df: pd.DataFrame) ->
    None: ...
```

Q1.5 Program-Major Balanced Cohort [3 marks]

- Using Sampling Strategy A, select 60 students such that all valid (program, major) combinations are represented approximately equally. [1]
- Show counts and heatmap. [1]
- Were any groups too small? How did you handle it? [1]

```
def sample_program_major_balanced(self, n: int) -> pd.DataFrame: ...
def show_program_major_counts_and_heatmap(self, sampled_df: pd.
   DataFrame) -> None: ...
```

Q2.0 k-Nearest Neighbors [30 marks]

Use k-NN (from sklearn) to predict `gender` based on student features. First, implement the following helper class for feature transformations.

```
class PerFeatureTransformer:
    def __init__(self):
        """Initializes memory for per-feature transformers."""
        ...
```

```
    def fit(self, df: pd.DataFrame, params: dict[str, str]) -> None:
        """Fits transformers for each feature based on the given
           type.
        Parameters:
            df: The dataframe containing features to be transformed.
            params: A dictionary mapping feature name to
                transformation type,
            e.g., {"GPA": "standard", "major": "ordinal", "program":
                "onehot"}.
        """
        ...

    def transform(self, df: pd.DataFrame) -> np.ndarray:
        """Applies the fitted transformers to the corresponding
           features and returns a NumPy array."""
        ...

    def fit_transform(self, df: pd.DataFrame, params: dict[str, str
       ]) -> np.ndarray:
        """Fits and transforms all features in one step using the
           given transformation parameters."""
        ...
```

Now, implement the following class for predicting gender using KNN.

```
class KNNGederPredictor:
    def __init__(self, student_df: pd.DataFrame, username: str):
        """Initializes the predictor with the full student dataset.
           Use the username for plots."""
        ...

    def train_val_test_split(self, test_size: float = 0.2, val_size:
        float = 0.2, seed: int = 42) -> tuple[pd.DataFrame, pd.
       DataFrame, pd.DataFrame]:
        ...
    def get_feature_matrix_and_labels(self, df: pd.DataFrame,
       features: list[str]) -> tuple[np.ndarray, np.ndarray]:
        """
        Extract selected features and gender labels from the
           DataFrame.
        Applies encoding to categorical variables and normalizes
           numeric features. Do not fit encoders or scalers on test
           data. Only transform using previously fitted ones.
        """
```

```
    def get_knn_accuracy_vs_k(self, k_values: list[int], distance:
       str = "euclidean") -> list[float]:
        """Calculates accuracy scores for various k values on the
           validation set."""
        ...

    def plot_knn_accuracy_vs_k(self, k_values: list[int], distance:
       str = "euclidean") -> None:
        """Plots accuracy scores against k values on the validation
           set."""
        ...

    def get_knn_f1_heatmap(self, k_values: list[int], distances:
       list[str]) -> pd.DataFrame:
        """Returns a dataframe with the f1-score for each
           combination on the validation set"""
        ...

    def plot_knn_f1_heatmap(self, f1_scores_df: pd.DataFrame) ->
       None: ...

    def get_knn_f1_single_feature_table(self, k_values: list[int],
       features: list[str], distance: str = "euclidean") -> pd.
       DataFrame:
        """Creates a table of F1 scores on the test set using only a
            single feature for prediction."""
        ...
```

Perform the following tasks.

- Train/val/test split the dataset and apply the data transforms. [4]
- What value of k (odd values from 1 to 21) gave the highest accuracy on the validation set with Euclidean distance metric? Justify with a plot. [2]
- Repeat the above for distance metrics like Manhattan and Cosine Similarity. [4]
- Report the validation F-1 score vs k for all the three distance metrics. [4]

- Plot a heatmap: $k \times$ distance function vs F-1 score. [4]
- Which distance metric performs better? Why might that be? [2]
- Instead of using all student features, an alternative is to use a single feature for prediction. Create an F-1 score table where rows are various values of k, columns are the single features used. Report values for test set for all the distance metrics. [6]
- Which single feature performed the best? How does it compare with the result using all the features? Why? [4]

Q3.0 Linear Regression with Regularization [30 marks]

You will predict `GPA` using student features. Use a validation set to select the hyperparameters.
Start with a function of the following form:

```
def run_poly_regression(X_train, y_train,
                        X_val, y_val,
                        X_test, y_test,
                        degree=1,
                        regularizer=None,
                        reg_strength=0.0):
    """
    Fit a polynomial regression model with optional regularization.

    Parameters:
        degree (int): Degree of the polynomial to fit
        regularizer (str or None): 'l1', 'l2', or None
        reg_strength (float): Regularization coefficient (alpha)

    Returns:
        dict with train, val, and test MSEs, and learned
            coefficients
    """
```

Perform the following tasks.

- For three setups - no regularization, L1 and L2 regularization, repeat the below steps: [8×3=24]
  - Fit polynomial regression models across degrees 1 to 6 [2]
  - Plot polynomial degree vs MSE (on train and validation sets). Describe the trend you observe as degree increases. [3]
  - For each degree, use val MSE to choose the best regularization strength. [1]
  - Plot regularization strength (log scale) vs val MSE for best degree. [2]

- Comment on performance improvement (if any) from regularization. Which overall experimental setup (degree, regularizer) yielded the best test performance? [3]
- For the best setup using L1 regularization, which features had non-zero weights? List the most important predictors for GPA. Repeat the same with L2 regularization. Comment on the differences. [3]