DSBDA Lab Assignments Implementation Guide

This guide covers step-by-step implementations for all DSBDA lab assignments. For each assignment, we state the objective, list required libraries/setup, show annotated code steps, interpret the expected output, and note observations (e.g. performance, data insights). Citations from relevant documentation and tutorials are included for clarity.

Assignment 1: Data Wrangling I

Objective: Perform basic data wrangling to prepare a dataset for analysis. This includes loading data, inspecting its structure, handling missing values, correcting data types, and encoding categorical variables. Data wrangling (aka data munging) involves transforming raw data into a clean, usable format.

Libraries & Setup: Use Python with Jupyter Notebook. Required libraries include:

pandas and numpy for data manipulation and numerical operations.

(Optional) scikit-learn if encoding categorical data or scaling is needed.

```
# Install libraries if needed (run once)
!pip install pandas numpy scikit-learn

# Import libraries
import pandas as pd
import numpy as np
```

Implementation Steps:

1. Load dataset: Read a CSV (or other) file into a pandas DataFrame.

```
df = pd.read_csv('data.csv')  # replace with actual file path
```

Explanation: pd.read_csv loads tabular data into df. This holds all columns and rows for further processing.

2. Inspect the data: Examine the first rows, summary info, and basic stats.

```
df.head()     # shows first few rows
df.info()     # shows columns, non-null counts, dtypes
df.describe() # summary statistics (mean, std, min, quartiles, max) for numeric cols
df.shape      # (number of rows, number of columns)
```

Explanation: df.head() gives sample rows, df.info() lists column names, data types, and non-null counts. df.describe() provides descriptive statistics (count, mean, std, min, quartiles, max) for numeric columns. This helps spot data quality issues.

3. Check and handle missing values: Identify missing entries and decide how to handle them.

```
df.isnull().sum()  # count missing per column
# Example: drop or fill missing values
df = df.dropna()   # drop rows with any nulls
# or
df['Age'] = df['Age'].fillna(df['Age'].mean())
```

Explanation: df.isnull().sum() reports the number of nulls in each column. Common strategies:

df.dropna() removes rows with any null values,

df.fillna(value) replaces nulls with a chosen value or statistic (e.g., mean).
This ensures subsequent analysis is not skewed by missing data.

4. Correct data types: Ensure each column has the appropriate type (integer, float, string, etc.).

```
df.dtypes  # view current data types
df['Salary'] = df['Salary'].astype(int)  # convert Salary from float/string to int
```

Explanation: Examine df.dtypes. If a numeric column was loaded as text (object), use astype(new_type) to convert. For example, df['col'] = df['col'].astype(int) converts a string of digits to integer. Proper types are necessary for analysis or modeling.

5. Encode categorical variables: Convert text categories into numeric form.

```
# One-hot encoding example for a 'Category' column
df = pd.concat([df, pd.get_dummies(df['Category'], prefix='Cat')], axis=1)
# or use label encoding
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
df['Category_Code'] = le.fit_transform(df['Category'])
```

Explanation: Machine learning models require numeric inputs. Use pd.get_dummies(df['col']) to create one-hot (binary) columns for each category. Alternatively, map categories to integers (e.g. df['category'].map({'A':0,'B':1})).

Expected Output: After these steps, df.head() should show no missing values in the cleaned columns, and all data types should be appropriate (e.g. numeric columns are int/float). Categorical columns will be represented numerically. For example, df.info() should report no nulls in handled columns and correct dtypes.

Observations: Inspect whether dropping missing rows significantly reduced data size or if fill values are sensible. Check distributions of imputed columns for anomalies. Encoding adds new columns – ensure no multicollinearity issues if using in modeling. Proper data wrangling usually improves model training stability and result interpretability.

Assignment 2: Data Wrangling II

Objective: Further clean the data by handling inconsistent values, detecting and treating outliers, and applying transformations or scaling. This refines the dataset for modeling and analysis.

Libraries & Setup: Same as Assignment 1, plus:

matplotlib or seaborn for visualizing (e.g., boxplots).

scikit-learn's StandardScaler for feature scaling.

```
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
```

Implementation Steps:

1. Handle inconsistent entries: Identify and correct typos or inconsistent categories (e.g., 'Yes' vs 'yes').

```
# Example: unify case or spelling
df['Gender'] = df['Gender'].str.strip().str.lower().map({'male':'M','female':'F'})
```

Explanation: Check for duplicate categories due to typos or case. Methods: string operations (strip, lower) or df['col'].replace({...}). Consistency ensures correct grouping.

2. Detect outliers: Use visualization or IQR method to spot extreme values.

```
# Boxplot for a numeric column, e.g. 'Age'
sns.boxplot(x=df['Age'])
plt.show()
# IQR method example
Q1 = df['Age'].quantile(0.25)
Q3 = df['Age'].quantile(0.75)
IQR = Q3 - Q1
# Filter out rows outside 1.5*IQR range
df = df[(df['Age'] >= Q1 - 1.5*IQR) & (df['Age'] <= Q3 + 1.5*IQR)]
```

Explanation: A boxplot (sns.boxplot) visually shows outliers as points beyond whiskers. Programmatically, compute Q1 and Q3 via df.quantile(), then IQR = Q3 - Q1. Exclude any df['col'] outside [Q1−1.5·IQR, Q3+1.5·IQR]. This is a standard rule to filter extreme outliers.

3. Apply transformations: If a numeric feature is skewed, apply log or sqrt to normalize it.

```
# Example: log-transform a positively skewed column 'Income'
df['Income_log'] = np.log1p(df['Income'])  # log(Income + 1)
```

Explanation: Taking log or square root can reduce skewness. np.log1p(x) computes log(1+x) to handle zeros. After transformation, check the new distribution (e.g., sns.histplot(df['Income_log'])) to ensure more symmetry.

4. Scale features: Standardize features to zero mean and unit variance if needed.

```
scaler = StandardScaler()
df[['Age_std','Income_std']] = scaler.fit_transform(df[['Age','Income_log']])
```

Explanation: StandardScaler subtracts the mean and divides by the standard deviation for each feature. This is important for many ML models (e.g. SVM, k-NN) to ensure features contribute equally. (Note: StandardScaler assumes no extreme outliers, as it is sensitive to them.)

Expected Output: Post-cleaning, visualizations (boxplots, histograms) should show removed or reduced outliers and more normal distributions if transformations were applied. For example, sns.histplot(df['Income_log']) should appear more symmetric than original. Check that scaled columns have mean ≈0 and std ≈1 (df[['Age_std','Income_std']].mean() and .std()).

Observations: Removing outliers may discard valuable data – ensure removed points are truly anomalies. After scaling, relationships between variables may change (e.g. correlation). Monitor whether transformations improved model predictions or interpretation. Note any significant loss of data from outlier removal.

Assignment 3: Descriptive Statistics

Objective: Compute summary statistics to understand the data's central tendencies and dispersions, both overall and within groups. This includes means, medians, variances, etc., often stratified by a categorical variable.

Libraries & Setup: Use:

pandas and numpy for statistics.

(Optional) matplotlib/seaborn to plot distributions or group comparisons.

```
import pandas as pd
import numpy as np
```

Implementation Steps:

1. Basic summary statistics: Use df.describe() and other aggregation functions.

```
df.describe()
# Specific stats:
df['Salary'].mean()
df['Salary'].median()
df['Salary'].std()
```

Explanation: df.describe() yields count, mean, std, min, quartiles, max for numeric columns. Alternatively, compute metrics individually (e.g., np.mean(df['col']), np.std(df['col'])). These give a quick sense of central tendency (mean/median) and variability (std).

2. Group-based statistics: If there is a categorical column (e.g. "Department"), compute stats within each group.

```
df.groupby('Category')['Value'].agg(['count','mean','median','std','min','max'])
```

Explanation: df.groupby('Category')['Value'].agg([...]) computes multiple statistics for each category level. This addresses "numeric variables grouped by a qualitative variable" scenario. For example, get mean and std of "Value" for each category of "Category".

3. Custom aggregations: Pandas also allows custom summary (e.g., quantiles or custom functions).

```
# Compute the 10th and 90th percentile of 'Score'
df['Score'].quantile([0.10, 0.90])
```

Explanation: Use df.quantile() or df.agg() with lambda for non-standard summaries. For instance, percentile values show outliers or tails.

Expected Output:

df.describe() should output a table like:

```
col1    col2  ...
count    ...     ...
mean   ...     ...
std    ...     ...
...
```

Grouped output (from groupby) yields a table with one row per category, e.g.:

```
count  mean  median  std    min    max
Category
A          50   1000   950    150   500   1500
B          45   1100  1050    130   600   1600
```

These summary tables let you interpret the data distribution. For example, compare means or medians across groups.

Observations: Check for differences between group statistics – e.g., one category may have much higher mean than others. Look at variance: a large std relative to the mean suggests high dispersion. If mean ≠ median, distribution may be skewed. These insights guide further analysis (e.g. whether to normalize or transform variables).

Assignment 4: Data Analytics I (Linear Regression)

Objective: Build and evaluate a simple linear regression model predicting a continuous target from one or more features. We fit a line to the data minimizing squared errors and evaluate goodness-of-fit (e.g., R^2).

Libraries & Setup:

scikit-learn for modeling: LinearRegression and metrics.

pandas, numpy for data handling.

(Optional) matplotlib/seaborn for plotting regression line.

```
import pandas as pd
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
import matplotlib.pyplot as plt
```

Implementation Steps:

1. Load the data: Read a dataset (e.g., Boston housing, custom data) into a DataFrame and define feature matrix X and target vector y.

```
df = pd.read_csv('housing.csv')  # example dataset
X = df[['feature1','feature2']].values  # replace with actual feature names
y = df['target'].values
```

Explanation: X should be shaped (n_samples, n_features), and y is the continuous target. For single-feature regression, X = df[['feature']] (2D array) is needed.

2. Train-test split: Randomly split data to evaluate performance on unseen data.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

Explanation: train_test_split from scikit-learn creates training and testing sets (e.g., 70% train, 30% test) for fair model evaluation.

3. Train the linear model: Fit the LinearRegression model on training data.

```
model = LinearRegression()
model.fit(X_train, y_train)
```

Explanation: By default, LinearRegression fits an intercept and weights to minimize the residual sum of squares (RSS). The learned coefficients are stored in model.coef_ and model.intercept_ after training.

4. Make predictions: Predict target values on the test set.

```
y_pred = model.predict(X_test)
```

Explanation: predict applies the linear model: $\hat{y} = X \cdot w + b$.

5. Evaluate performance: Compute regression metrics, such as Mean Squared Error (MSE) and R^2.

```
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print(f"MSE: {mse:.2f}, R^2: {r2:.3f}")
```

Explanation: MSE = average squared error. R^2 (coefficient of determination) is defined as 1 - \frac{\sum(y_i - \hat y_i)^2}{\sum(y_i - \bar y)^2}. $R^2 = 1$ indicates perfect fit; $R^2 = 0$ means model is no better than predicting the mean. R^2 may be negative if model is worse.

6. Visualize results (optional): Plot the regression line against data points (for 1D regression).

```
plt.scatter(X_test[:,0], y_test, color='blue', label='Actual')
plt.scatter(X_test[:,0], y_pred, color='red', alpha=0.6, label='Predicted')
plt.xlabel('Feature')
plt.ylabel('Target')
plt.legend()
plt.show()
```

Explanation: A scatter plot of true vs. predicted values (or fit line) helps visually assess fit quality.

Expected Output: The printed metrics (MSE and R^2) summarize model performance. For example:

```
MSE: 24.53, R^2: 0.85
```

This indicates the regression explains 85% of variance ($R^2=0.85$). A high R^2 and low MSE show good fit. The scatter plot should show predictions lying close to actual values, ideally near a 45° line if plotting y_pred vs y_test.

Observations: Check residuals (y_test - y_pred) for patterns (should be random). If R^2 is low, the model may be missing important features or relationships (e.g. nonlinearity). Very high R^2 on train vs low on test indicates overfitting. Examine coefficient values: large magnitudes suggest strong influence of that feature.

Assignment 5: Data Analytics II (Logistic Regression)

Objective: Build a logistic regression classifier for binary outcomes. For example, predict whether a customer purchases (1) or not (0) based on features like age or salary. Evaluate via classification metrics (accuracy, confusion matrix).

Libraries & Setup:

pandas, numpy for data.

scikit-learn for LogisticRegression, train_test_split, and metrics (accuracy_score, precision_score, recall_score, confusion_matrix).

(Optional) matplotlib for plotting decision boundary if 2D data.

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_score
```

Implementation Steps:

1. Load data: Read a dataset (e.g. “Social_Network_Ads.csv”) into df. Define features X and binary target y.

```
df = pd.read_csv('Social_Network_Ads.csv')
X = df[['Age', 'EstimatedSalary']].values
y = df['Purchased'].values  # 0 or 1
```

Explanation: Here X includes relevant numeric features. Ensure y is 0/1 for purchased status.

2. Split data: Create training and test sets.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)
```

Explanation: As before, this ensures the model is evaluated on unseen data.

3. Train logistic model: Fit LogisticRegression on training data.

```
clf = LogisticRegression()
clf.fit(X_train, y_train)
```

Explanation: LogisticRegression fits a model that outputs probabilities. It maximizes likelihood for binary classification.

4. Predict on test data:

```
y_pred = clf.predict(X_test)
```

Explanation: For each test sample, predict returns 0 or 1 class labels.

5. Compute confusion matrix:

```
cm = confusion_matrix(y_test, y_pred)
print("Confusion matrix:\n", cm)
```

Explanation: A confusion matrix C is such that counts samples known to be in class i and predicted as class j. For binary,

True Negatives = C[0,0], False Positives = C[0,1]

False Negatives = C[1,0], True Positives = C[1,1].

6. Calculate metrics: Accuracy, precision, and recall.

```
acc = accuracy_score(y_test, y_pred)
prec = precision_score(y_test, y_pred)
rec = recall_score(y_test, y_pred)
print(f"Accuracy: {acc:.2f}, Precision: {prec:.2f}, Recall: {rec:.2f}")
```

Explanation:

Accuracy: (TP + TN) / (total).

Precision: TP / (TP + FP) – proportion of positive predictions that are correct.

Recall (Sensitivity): TP / (TP + FN) – proportion of actual positives detected.
These quantify classifier performance. (Equations implied by definitions of TP, FP, FN from the matrix.)

Expected Output: Example printouts:

```
Confusion matrix:
 [[65  5]
 [ 8 42]]
Accuracy: 0.88, Precision: 0.89, Recall: 0.84
```

Interpretation: of 70 negatives, 65 correctly predicted (5 FP); of 50 positives, 42 correctly predicted (8 FN). Accuracy=(65+42)/120≈0.88. Precision=42/(42+5)=0.89, Recall=42/(42+8)=0.84. These indicate solid performance.

Observations: If precision >> recall, the model is conservative (few false positives); if recall >> precision, it catches most positives but with more false alarms. A balanced F1-score might be considered. Inspect misclassified cases: are there patterns (e.g. certain age groups mispredicted)? Adjust model (e.g. add features or regularization) if needed.

Assignment 6: Data Analytics III (Naïve Bayes)

Objective: Apply the Gaussian Naive Bayes classifier to a dataset (e.g. Iris) and evaluate its classification accuracy. GaussianNB assumes each feature is normally distributed and features are conditionally independent given the class.

Libraries & Setup:

pandas, numpy for data.

scikit-learn for GaussianNB, splitting, and metrics.

```
import pandas as pd
import numpy as np
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, accuracy_score
```

Implementation Steps:

1. Load data: Use the Iris dataset, available via sklearn.datasets or CSV.

```
from sklearn.datasets import load_iris
iris = load_iris()
X = iris.data   # sepal/petal lengths and widths
y = iris.target # species labels (0,1,2)
```

Explanation: X has four continuous features. GaussianNB assumes each follows a Gaussian distribution.

2. Train-test split: (Optional since Iris is small; can use all for training)

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)
```

3. Train Naive Bayes model:

```
nb = GaussianNB()
nb.fit(X_train, y_train)
```

Explanation: GaussianNB learns class priors and class-conditional means/variances for each feature.

4. Predict and evaluate:

```
y_pred = nb.predict(X_test)
cm = confusion_matrix(y_test, y_pred)
acc = accuracy_score(y_test, y_pred)
print("Confusion matrix:\n", cm)
print(f"Accuracy: {acc:.2f}")
```

Explanation: The confusion matrix (3×3) shows correct classifications along diagonal. Accuracy = proportion of correctly classified samples.

Expected Output: For Iris, typical accuracy ~0.93-0.97. Example:

```
Confusion matrix:
 [[16  0  0]
 [ 0 15  1]
 [ 0  0 13]]
Accuracy: 0.97
```

Interpretation: Out of 45 test samples, 44 correctly classified, 1 misclassified (an Iris-versicolor as Iris-virginica).

Observations: GaussianNB often performs well on Iris with few samples. The confusion matrix diagonal entries reflect true positive counts per class. Compare classes: if one class has many errors, features may overlap for that class. Note that Naive Bayes provides probabilities (predict_proba), but its probabilistic predictions may not be well-calibrated (it is a good classifier but not a precise probability estimator). The accuracy here indicates very high performance on this simple dataset.

Assignment 7: Text Analytics

Objective: Perform basic text preprocessing and feature extraction on text data. Steps include tokenization, POS tagging, stop-word removal, stemming, lemmatization, and computing term frequencies (TF) and TF-IDF features.

Libraries & Setup:

NLTK for NLP tasks: word_tokenize, pos_tag, stopwords, PorterStemmer, WordNetLemmatizer.

scikit-learn for text vectorization: CountVectorizer, TfidfVectorizer.

```
import nltk
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
nltk.download('stopwords')
nltk.download('wordnet')

from nltk.tokenize import word_tokenize
from nltk import pos_tag
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer, WordNetLemmatizer

from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
```

Implementation Steps:

1. Tokenization: Split raw text into tokens (words/punctuation).

```
text = "NLTK is a leading platform for building Python programs to work with human language data."
tokens = word_tokenize(text)
print(tokens)
```

Explanation: word_tokenize breaks a sentence into words/punctuation. E.g. "NLTK is a leading platform..." → ['NLTK', 'is', 'a', 'leading', 'platform', ...].

2. POS Tagging: Label each token with its part-of-speech.

```
pos_tags = pos_tag(tokens)
print(pos_tags)
```

Explanation: pos_tag returns tuples like ('platform', 'NN'), identifying nouns, verbs, etc. Knowing POS can improve later steps (e.g. correct lemmatization as verb vs noun).

3. Remove stop words: Filter out common words with little meaning.

```
stop_words = set(stopwords.words('english'))
filtered_tokens = [t for t in tokens if t.lower() not in stop_words]
print(filtered_tokens)
```

Explanation: NLTK's English stopword list includes words like “the”, “is”, “and”. Removing them focuses analysis on meaningful terms. Here, ['NLTK', 'leading', 'platform', 'building', 'Python', ...] remains.

4. Stemming: Reduce tokens to root form using Porter's algorithm.

```
stemmer = PorterStemmer()
stems = [stemmer.stem(t) for t in filtered_tokens]
print(stems)
```

Explanation: Stemming chops off word endings (e.g. “running” → “run”). It is a crude form of normalization. Porter stemmer is a common choice.

5. Lemmatization: Convert words to their lemma (dictionary form).

```
lemmatizer = WordNetLemmatizer()
lemmas = [lemmatizer.lemmatize(t) for t in filtered_tokens]
print(lemmas)
```

Explanation: Lemmatization is more sophisticated: it uses vocabulary and morphology to find the base form. E.g., “dogs” → “dog”, “running” → “run”. Unlike stemming, it yields real words as output. (Specifying POS tags yields better results, e.g. lemmatizer.lemmatize('running','v').)

6. Term Frequency (TF): Count term occurrences in one or more documents.

```
corpus = [
   "I love data science",
   "Data science and machine learning"
]
cv = CountVectorizer()
tf_matrix = cv.fit_transform(corpus)
print(cv.get_feature_names_out(), tf_matrix.toarray())
```

Explanation: CountVectorizer tokenizes and counts word frequencies. For the corpus, vocabulary might be ['and','data','learning','love','machine','science'], and the array shows counts of each word per document.

7. TF-IDF: Compute TF-IDF scores to weight terms by rarity.

```
tv = TfidfVectorizer()
tfidf_matrix = tv.fit_transform(corpus)
print(tv.get_feature_names_out(), tfidf_matrix.toarray())
```

Explanation: TF-IDF multiplies term frequency by inverse document frequency, giving higher weight to terms that are frequent in a document but rare across documents. The formula is IDF = log(N/(1+df)) and TF-IDF highlights important words.

Expected Output:

Tokens and POS: A list of tokens (words) and a list of (token, POS) pairs.

Filtered tokens: List without common stopwords.

Stems vs lemmas: Lists showing root forms (e.g. ['nl', 'lead', 'platform', 'build', 'python', ...] vs ['NLTK', 'leading', 'platform', 'building', 'Python', ...]).

CountVectorizer: Feature names and count arrays, e.g.:

```
['and','data','learning','love','machine','science']
[[0 1 0 1 0 1]
 [1 1 1 0 1 1]]
```

TF-IDF: Numeric array of TF-IDF scores. Each row sums to at most ~1.

Observations:

After removing stopwords, key terms stand out. For instance, “NLTK”, “Python”, “platform”, “data”, “science” remain.

Stemming/lemmatization will generally reduce word variety (e.g. “data” stays “data”, “learning”→“learn”).

TF counts highlight the most common words; TF-IDF downweights words that appear in both docs (“data”, “science” might have lower TF-IDF if in all docs) and upweights unique terms like “love”.

Understanding these transformations is crucial for text classification or clustering tasks.

Assignment 8: Data Visualization I

Objective: Explore data using visualizations. For the Titanic dataset (passenger data), we will plot relationships to discover patterns. Example tasks: scatterplots, heatmaps, and distribution plots.

Libraries & Setup:

seaborn and matplotlib for plotting (Seaborn’s high-level plots are convenient).

```
import seaborn as sns
import matplotlib.pyplot as plt
sns.set_theme(style="whitegrid")
```

Implementation Steps:

1. Load Titanic data: Use Seaborn’s built-in dataset.

```
titanic = sns.load_dataset('titanic')
titanic.head()
```

Explanation: This dataframe includes passenger features like age, sex, class, fare, and survival (0/1).

2. Pairplot for feature relationships:

```
sns.pairplot(titanic.dropna(subset=['age','fare','survived']),
             vars=['age','fare','pclass'], hue='survived')
plt.show()
```

Explanation: sns.pairplot shows scatterplots of each numeric pair and histograms on the diagonal. Using hue='survived' colors points by survival status. This helps visualize any separation between survivors vs. non-survivors across features.

3. Correlation heatmap:

```
corr = titanic[['age','fare','pclass','survived']].corr()
sns.heatmap(corr, annot=True, cmap='coolwarm')
plt.show()
```

Explanation: Heatmap of the correlation matrix highlights linear relationships. For example, pclass (ticket class) might correlate negatively with fare (higher class = higher fare) or with survival rate.

4. Histogram of fares:

```
sns.histplot(titanic['fare'].dropna(), bins=20, kde=True)
plt.xlabel('Fare')
plt.title('Distribution of Ticket Fare')
plt.show()
```

Explanation: The histogram with a KDE curve shows how ticket prices are distributed. It reveals skewness (often right-skewed: many low fares, few very high). The KDE (smoothed curve) helps see the overall shape.

5. Interpretation: Look for skewness and central tendency. For instance, if most passengers paid low fares and a few paid very high fares, this skew is visible.

Expected Output:

Pairplot: A grid of scatterplots (e.g., age vs fare) colored by survival. If survivors cluster differently (e.g., younger ages?), patterns appear.

Heatmap: A matrix (e.g., fare vs pclass has strong correlation, survived vs sex maybe not shown here but could).

Histogram: Bars showing counts of passengers in fare ranges. Likely a peak at lower fares (most tickets cheap), tail at high fares. The KDE shows a single peak near low fare and a long tail.

Observations:

The fare histogram often shows that most passengers paid low fares, with skew (few outliers paying very high). One might note that class (pclass) correlates with fare (first-class paid more).

In the pairplot or heatmap, high survival might correlate with different features: e.g., women and children survived more (not visualized here).

These plots guide further questions: e.g., “Did higher fare (rich class) get higher survival?” If so, we expect a correlation between fare and survived. A future assignment might examine that directly.

Assignment 9: Data Visualization II

Objective: Create comparative visualizations to analyze distributions across categories. Continuing with Titanic data, we will compare age distributions by sex and survival status using boxplots.

Libraries & Setup: Same as above (seaborn, matplotlib).

```
import seaborn as sns
import matplotlib.pyplot as plt
sns.set_theme(style="ticks")
```

Implementation Steps:

1. Load data: Use the same titanic DataFrame from Assignment 8.

2. Boxplot of age by sex and survival:

```
sns.boxplot(x='sex', y='age', hue='survived', data=titanic)
plt.title('Age distribution by Sex and Survival')
plt.show()
```

Explanation: sns.boxplot creates side-by-side boxplots for each sex (male/female), with separate boxes for survival=0 vs 1 (indicated by hue). Each box shows median line and interquartile range; whiskers and outliers beyond them appear as dots.

3. Display the plot: Ensure plt.show() renders it (as above).

4. Analyze results: Examine the medians and spreads. For example, compare the median age of surviving vs non-surviving females and males.

Explanation: The boxplot reveals if one group had consistently higher ages or more variability. Outliers (dots) beyond whiskers are visible, indicating unusual ages (very old or very young passengers). For instance, one might note: “Female survivors tend to have lower median age than male survivors” or “non-survivor outliers are few”.

Expected Output: A plot with four boxes: Female-survived, Female-not, Male-survived, Male-not. Each box shows age distribution. Example observations:

Females (blue) may have higher survival and perhaps a tighter age range.

Males (orange) might show lower survival rates (if the hue colors indicate that).

The median lines allow quick comparison of typical ages between groups.

Observations:

If the median age of survivors is lower/higher, it suggests age played a role.

Compare variance: e.g., if non-survivor females have a wider box, their ages were more spread.

Presence of outliers (dots) shows any extremely old/young passengers.

This helps answer: “Did young passengers survive more?” If so, young (small age) survivors might dominate the survived boxes.

These inferences guide feature selection or hypothesis for modeling (e.g., including Age and Sex as predictors in a survival model).

Assignment 10: Data Visualization III

Objective: Explore feature distributions within a dataset. Using the Iris dataset, plot histograms and boxplots for each numeric feature to understand their ranges, shapes, and outliers, which helps in comparing species distributions.

Libraries & Setup:

seaborn and matplotlib.

```
import seaborn as sns
import matplotlib.pyplot as plt
sns.set_theme(style="whitegrid")
```

Implementation Steps:

1. Load Iris data:

```
iris = sns.load_dataset('iris')
iris.info()  # to confirm features and types
```

Explanation: Iris has four numeric features: sepal_length, sepal_width, petal_length, petal_width, and a categorical species. All four are floats (verified by info()).

2. Histogram for each feature:

```
features = ['sepal_length','sepal_width','petal_length','petal_width']
for col in features:
    plt.figure()
    sns.histplot(iris[col], bins=15, kde=False)
    plt.title(f"Histogram of {col}")
    plt.show()
```

Explanation: This loops over features, plotting a histogram of values. The shape (e.g., roughly normal or bimodal) can indicate how data is distributed. For instance, petal_length often shows two peaks (different species), whereas sepal_width might be unimodal.

3. Boxplot for each feature:

```
for col in features:
   plt.figure()
   sns.boxplot(y=iris[col])
   plt.title(f"Boxplot of {col}")
   plt.show()
```

Explanation: Each boxplot shows median, quartiles, and outliers for that feature. Outliers (points beyond whiskers) indicate unusual values. For example, petal_length may have no extreme outliers, whereas sepal_width might.

4. Compare distributions: Analyze which features have skew or outliers.
Explanation: After plotting, note differences. Perhaps petal length and width separate species well (e.g., Setosa has much smaller petal widths than others, possibly visible as a distinct cluster or range).

Expected Output:

Histograms: Four charts. Example: petal_length histogram shows a bimodal distribution (Iris-setosa vs others). sepal_length might look more normal with slight skew.

Boxplots: Four separate boxes. E.g., petal_width may have minimal outliers but different medians for the cluster of values. The boxplot of petal_length might show some outliers (rare long petals).

Observations:

Identify outliers: any points beyond whiskers are flagged for inspection (e.g., perhaps data entry errors or rare species measurements).

Compare ranges: e.g., sepal_length ranges ~4.3–7.9 cm, petal_length ~1.0–6.9 cm. Petal measurements generally have higher variance.

These insights help in classification: e.g., Iris-setosa often has much smaller petal sizes – the histograms may show two clusters. Recognizing such patterns informs feature importance in modeling.

Assignment 11: Hadoop Word Count

Objective: Implement the classic Word Count MapReduce job using Hadoop Streaming with Python scripts. The goal is to count the frequency of each word in a large text file (e.g., a log file).

Libraries & Setup: No special Python libraries needed beyond standard I/O. Set up Hadoop (or Hadoop streaming). Prepare two executable Python scripts: mapper.py and reducer.py. Ensure they have UNIX executable permissions (chmod +x).

Implementation Steps:

1. Write the mapper script (mapper.py): It reads lines from standard input, splits into words, and outputs each word with a count of 1.

```
#!/usr/bin/env python3
import sys
for line in sys.stdin:
    line = line.strip()
    words = line.split()
    for word in words:
        print(f"{word}\t1")
```

Explanation: For each input line, we remove whitespace, split by space, and emit <word>\t1. This is the key-value format expected by Hadoop Streaming. Each word occurrence generates a line like "error\t1".

2. Write the reducer script (reducer.py): It reads sorted key-value pairs from stdin, aggregates counts per word, and emits the total.

```
#!/usr/bin/env python3
import sys

current_word = None
current_count = 0
for line in sys.stdin:
    line = line.strip()
    word, count = line.split('\t', 1)
    count = int(count)
    if current_word == word:
        current_count += count
    else:
        if current_word:
            print(f"{current_word}\t{current_count}")
        current_word = word
        current_count = count
# output the count for the last word
if current_word:
    print(f"{current_word}\t{current_count}")
```

Explanation: The Hadoop framework (or manual pipeline) must sort mapper output by key. The reducer loops through each line: if the word is the same as the previous, it accumulates; otherwise, it outputs the previous word's total and starts a new count. This sums all counts for each word.

3. Run the job (Hadoop streaming): Using a Hadoop cluster, you would run something like:

```
hadoop jar /path/hadoop-streaming.jar \
  -mapper /path/mapper.py \
  -reducer /path/reducer.py \
  -input /path/input.txt \
  -output /path/output_dir
```

Explanation: This tells Hadoop to use the Python scripts as mapper and reducer. (Alternatively, to test locally: cat input.txt | python mapper.py | sort | python reducer.py.)

4. Interpret results: The output in HDFS (or console) will be lines of word<TAB>count.
Explanation: Each line like error	13 means the word "error" appeared 13 times in the input. The final output is the frequency of each word in the dataset.

Expected Output: A list of words with counts, e.g.:

```
a	150
data 75
error 13
```

(sorted by word). The exact output depends on input text. One should verify sanity: total counts should sum to number of word tokens. For a simple input "hello world hello", expect:

~~~~
hello    2
world    1
```.
~~~~

Observations:
- The Word Count program scales to very large datasets via MapReduce.
- The output can be used to identify the most common words (e.g., using a sort or max).
- In large data, rare words may appear only once or twice (long tail).
- Make sure to handle case sensitivity or punctuation if needed (e.g., convert words to lowercase in the mapper to count “Error” and “error” as same).

Assignment 12: Hadoop Weather Data

Objective: Develop a MapReduce job to process weather data. For example, compute the average (or maximum) annual temperature from large weather records using Hadoop Streaming.

Libraries & Setup: Like Word Count, use two Python scripts and Hadoop Streaming. Ensure data is preprocessed (CSV format) and scripts are executable.

Implementation Steps:
1. **Mapper function (`mapper_weather.py`):** Read each CSV line, parse the date to extract year, and emit (year, temperature). Example format: `YYYY-MM-DD,StationID,temp`.

~~~~
   ```python
   #!/usr/bin/env python3
   import sys
   for line in sys.stdin:
       line = line.strip()
       fields = line.split(',')
       if len(fields) < 3:
           continue
       date, location, temp = fields[0], fields[1], fields[2]
       year = date.split('-')[0]
       try:
           temp = float(temp)
       except ValueError:
           continue
       print(f"{year}\t{temp}")
~~~~

Explanation: For each record, extract year and numeric temp, then output year<TAB>temperature. This prepares values by year for aggregation.

2. Reducer function (reducer_weather.py): Aggregate temperatures per year. Here we compute average; alternatively, compute max by tracking the maximum value.

```
#!/usr/bin/env python3
import sys

current_year = None
temps = []
for line in sys.stdin:
    line = line.strip()
    year, temp = line.split('\t', 1)
    temp = float(temp)
    if current_year == year:
        temps.append(temp)
    else:
        if current_year:
            avg_temp = sum(temps) / len(temps)
            print(f"{current_year}\t{avg_temp:.2f}")
        current_year = year
        temps = [temp]
if current_year:
    avg_temp = sum(temps) / len(temps)
    print(f"{current_year}\t{avg_temp:.2f}")
```

Explanation: As lines come sorted by year, accumulate all temp values in a list. When the year changes, compute average and emit (year, avg_temp). (For maximum, keep a running max instead of a list.)

3. Run with Hadoop Streaming:

```
hadoop jar .../streaming.jar \
  -mapper /path/mapper_weather.py \
  -reducer /path/reducer_weather.py \
  -input /weather_data.csv \
  -output /weather_output
```

Explanation: Similar to Word Count. The mapper and reducer handle key (year) and values as shown.

4. Interpret results: Output lines like YYYY   value.
Explanation: For average temperature task, line 2010   58.32 means the average temperature in 2010 was 58.32. If using max, it's the max for that year. This summarizes each year's data.

Expected Output: A list of years with computed climate statistic, e.g.:

```
2018    72.45
2019    73.10
2020    71.85
```

(Values depend on input data.)

Observations:

Shows yearly trends in weather (e.g., rising temps).

If average is used, outliers (e.g., one extremely hot day) have limited influence; with max, highlights extremes (maybe heatwaves).

Ensures parallel processing: each node can handle a subset of data.

Check data quality: missing or erroneous values are skipped by mapper, so reducer counts only valid temps.

References: We followed the standard MapReduce design for Word Count and weather aggregation, using Hadoop Streaming with Python mapper/reducer scripts. Each output line (key, value) pair (e.g. ("hello", 13) or ("2010", 58.32)) summarizes the aggregated result for that key.