Data Science & Big Data Analytics Lab – Assignment Explanations

Assignment 1: Data Wrangling I

- **Import libraries:** Load packages like `pandas` and `numpy` using `import pandas as pd`, `import numpy as np`. These provide data structures and numerical tools for analysis.
- **Load dataset:** Use `pd.read_csv('filename.csv')` to read an open-source CSV file into a pandas DataFrame (`df`). This holds the tabular data.
- **Inspect data:** Call `df.head()` or `df.info()` to view the first rows and summary information (column names, types, non-null counts). Use `df.describe()` to get summary statistics (mean, std, min, max) for numeric columns. Check `df.shape` for dimensions.
- **Check missing values:** Use `df.isnull().sum()` to count missing entries in each column. Handle missing data with methods like `df.dropna()` (remove rows with nulls) or `df.fillna(value)` (replace nulls with a value or mean).
- **Adjust data types:** Examine `df.dtypes` to see each column's type (integer, float, object for strings, etc.). Convert types if needed using `df['col'] = df['col'].astype(new_type)` (for example, convert a numeric string to integer).
- **Encode categorical variables:** For any categorical columns, create numeric representations. For example, use `pd.get_dummies(df['category'])` to one-hot encode text categories into binary columns, or map text labels to numbers with `df['category'].map({'A':0,'B':1})`. This turns categories into numeric form for analysis.

Assignment 2: Data Wrangling II

- **Handle missing/inconsistent data:** Similar to Assignment 1, check `df.isnull().sum()` and use methods like `df.dropna()` or `df.fillna()` to deal with missing entries. If there are inconsistent values (e.g. typos), correct them or fill with appropriate values.
- **Detect and treat outliers:** Identify outliers in numeric columns. For example, use `df.boxplot()` or calculate the interquartile range (IQR) with `Q1 = df['col'].quantile(0.25)`, `Q3 = df['col'].quantile(0.75)`, `IQR = Q3 - Q1`. Remove rows where values lie beyond `[Q1 - 1.5*IQR, Q3 + 1.5*IQR]`. This filters out extreme values.
- **Apply data transformations:** If a variable is skewed or on an inconvenient scale, transform it. For example, apply `np.log(df['col'])` or `np.sqrt(df['col'])` to reduce skewness. After transformation, replace or add columns: `df['col_log'] = np.log(df['col'])`. Optionally, normalize or standardize data using `StandardScaler` from `sklearn` to rescale features.

Assignment 3: Descriptive Statistics

- **Group-based statistics:** Use pandas `groupby()` to compute statistics by category. For example, `df.groupby('Category')['Value'].agg(['mean','median','min','max','std'])` gives mean, median, min, max, and standard deviation of *Value* within each *Category* group. This addresses "numeric variables grouped by a qualitative variable."

- **Basic summary stats:** Use `df.describe()` for an overall summary (count, mean, std, min, quartiles, max) of all numeric columns. Alternatively, apply `np.mean()`, `np.median()`, `np.std()` on specific columns. These functions yield central tendency and dispersion measures.

- **Percentiles and detailed stats:** To get percentiles, use numpy or pandas. E.g., `np.percentile(iris_df['sepal_length'], [25,50,75])` gives the 25th, 50th (median), and 75th percentiles of sepal length in the Iris set. Libraries like `scipy.stats.describe(iris_df['sepal_length'])` can also return count, min, max, mean, variance, and percentiles, fulfilling the task to display detailed statistics of Iris species.

Assignment 4: Data Analytics I (Linear Regression)

- **Load Boston Housing data:** Read the dataset (e.g. from Kaggle or via `sklearn.datasets.load_boston()`) into a DataFrame. Separate features matrix `X` (all columns except price) and target vector `y` (house prices).

- **Split data:** Use `from sklearn.model_selection import train_test_split` to divide data into training and testing sets, e.g. `X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)`.

- **Train linear regression model:** Import `LinearRegression` from `sklearn.linear_model` and create a model: `model = LinearRegression()`. Fit it using `model.fit(X_train, y_train)`. This finds the best-fit line coefficients.

- **Make predictions and evaluate:** Predict prices on test set with `y_pred = model.predict(X_test)`. Evaluate performance by computing R^2 (`model.score(X_test, y_test)`) or mean squared error (`from sklearn.metrics import mean_squared_error; mean_squared_error(y_test, y_pred)`).

- **Interpret coefficients:** Access `model.coef_` to see how each feature affects price (positive or negative influence), and `model.intercept_` for the baseline. Understanding these values explains the model in simple terms.

Assignment 5: Data Analytics II (Logistic Regression)

- **Load Social Network data:** Read `Social_Network_Ads.csv` into `df`. Define feature matrix `X` (e.g. columns like `Age`, `EstimatedSalary`) and target `y` (purchased or not).

- **Split into train/test:** Use `train_test_split` as before to create training and testing sets.

- **Train logistic model:** Import `LogisticRegression` (`from sklearn.linear_model`) and fit: `clf = LogisticRegression(); clf.fit(X_train, y_train)`. This trains a classification model.

- **Predict on test data:** Compute `y_pred = clf.predict(X_test)`. This predicts purchase (0 or 1) for each test example.

- **Confusion matrix:** Generate the confusion matrix with `from sklearn.metrics import confusion_matrix; cm = confusion_matrix(y_test, y_pred)`. This yields counts of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN).

- **Compute evaluation metrics:** From the confusion matrix, calculate accuracy (`(TP+TN)/(TP+TN+FP+FN)`), precision (`TP/(TP+FP)`), recall (`TP/(TP+FN)`), etc. Alternatively, use `accuracy_score`, `precision_score`, `recall_score` from `sklearn.metrics` on `y_test` and `y_pred`. These values describe model performance in simple terms.

Assignment 6: Data Analytics III (Naïve Bayes)

- **Load Iris dataset:** Use `pd.read_csv('iris.csv')` or `sklearn.datasets.load_iris()` to get data. Let `X` be the feature columns (sepal/petal sizes) and `y` the species labels.
- **Train-test split:** Optionally split data if evaluating performance.
- **Train Naïve Bayes model:** Import `GaussianNB` (`from sklearn.naive_bayes`). Instantiate and fit: `nb = GaussianNB(); nb.fit(X_train, y_train)`. This fits the probabilistic model.
- **Predict and evaluate:** Use `y_pred = nb.predict(X_test)` to classify test samples. Again compute `confusion_matrix(y_test, y_pred)` and derive accuracy, precision, recall.
- **Result interpretation:** The outputs show how well the Naïve Bayes model classifies Iris species. Mentioning accuracy or confusion matrix entries helps understand its success.

Assignment 7: Text Analytics

- **Tokenization:** Break a text document into tokens (words) using NLTK: `from nltk.tokenize import word_tokenize; tokens = word_tokenize(text)`. This splits sentences into a list of words/punctuation.
- **POS Tagging:** Tag each token with its part of speech: `from nltk import pos_tag; pos_tags = pos_tag(tokens)`. This identifies nouns, verbs, etc. by labeling each token.
- **Remove stop words:** Filter out common words (stopwords) that carry little meaning. Use `from nltk.corpus import stopwords`; then `tokens = [t for t in tokens if t.lower() not in stopwords.words('english')]`. This leaves only the significant words.
- **Stemming:** Reduce words to their root form using Porter's algorithm: `from nltk.stem import PorterStemmer; stemmer = PorterStemmer(); stems = [stemmer.stem(t) for t in tokens]`. For example, "running" becomes "run".
- **Lemmatization:** Convert words to dictionary form (lemmas): `from nltk.stem import WordNetLemmatizer; lemmatizer = WordNetLemmatizer(); lemmas = [lemmatizer.lemmatize(t) for t in tokens]`. This also often yields "run" from "running", considering POS if provided.
- **Term Frequency (TF):** Count how often each word appears in the document. You can manually count (e.g., using `collections.Counter(tokens)`), or use `sklearn.feature_extraction.text.CountVectorizer` to convert text to a matrix of token counts.
- **Inverse Document Frequency (IDF):** Weight terms by how rare they are across documents. Use `TfidfVectorizer` from sklearn, which computes TF-IDF automatically. Conceptually, `IDF = log(N / (1 + df))` where `df` is document frequency. Multiplying TF by IDF gives TF-IDF, highlighting important words.

Assignment 8: Data Visualization I

- **Load Titanic dataset:** Use Seaborn's built-in data: `import seaborn as sns; titanic = sns.load_dataset('titanic')`. This DataFrame has passenger info.

- **Explore relationships:** Create visualizations to find patterns. For example, use `sns.pairplot(titanic)` to see scatterplots of numeric features, or `sns.heatmap(titanic.corr(), annot=True)` to see correlations between variables. These highlight any obvious trends.

- **Plot ticket price distribution:** Visualize how fares are distributed. Use a histogram or density plot: e.g. `import matplotlib.pyplot as plt; sns.histplot(titanic['fare'], bins=20, kde=True)`. This shows the frequency of different fare amounts.

- **Interpret chart:** Look at the histogram to see if most fares are low/high, if distribution is skewed, etc. A smooth curve over the histogram (KDE) helps identify the general shape of distribution.

Assignment 9: Data Visualization II

- **Load Titanic data:** Continue with the same `titanic` DataFrame.

- **Boxplot of age by sex and survival:** Use Seaborn to compare age distributions: `sns.boxplot(x='sex', y='age', hue='survived', data=titanic)`. This draws side-by-side boxplots for male vs female, and further separates by survival (survived=0 or 1).

- **Display plot:** Call `plt.show()` to render the figure.

- **Analyze results:** The boxplot shows the median age line and the interquartile range for each subgroup. Outliers (dots beyond whiskers) are obvious. One can note, for example, if women had higher survival or if age differs between survivors and non-survivors.

Assignment 10: Data Visualization III

- **Load Iris dataset:** Read the Iris data into `iris` (e.g. `iris = sns.load_dataset('iris')`).

- **Feature list and types:** Check `iris.info()` or `iris.dtypes` to list features (`sepal_length`, `sepal_width`, `petal_length`, `petal_width`) and note that they are numeric (floats).

- **Histogram for each feature:** For each numeric column, plot its histogram. For example: `iris['sepal_length'].hist(bins=15)` or `sns.histplot(iris['sepal_length'], kde=False)`. Repeat for sepal/petal widths and lengths. This shows each feature's distribution (e.g. roughly normal or skewed).

- **Boxplot for each feature:** To illustrate spread and outliers, use boxplots. One approach is `iris.boxplot()` on numeric columns, or loop: `for col in iris.columns[:-1]: sns.boxplot(y=iris[col])`. Each box shows median and quartiles, with circles for outliers.

- **Compare distributions and outliers:** By examining the histograms and boxplots, identify which features have outliers (points beyond whiskers) or different ranges. For instance, petal length may show more separation between species. These visuals help infer data characteristics.

Assignment 11: Hadoop Word Count

- **Mapper design:** In a MapReduce job, the mapper reads each line of input (the log file) and splits it into words (e.g. `words = line.split()`). For each word, the mapper outputs a key-value pair `(word, 1)`. In code, it often prints `word\t1`. This assigns an initial count of 1 to every occurrence.

- **Combiner (optional) / Reducer design:** Hadoop first optionally aggregates mapper outputs. The reducer then processes sorted input by key (word). It sums all counts for each word: e.g. if inputs are `("hello",1),`

`("hello",1), ...`, the reducer adds them up to get `("hello", total_count)`. This is typically implemented by looping over values and accumulating a sum.

- **Running the job:** If using Hadoop streaming with Python scripts, the command includes `-mapper mapper.py` and `-reducer reducer.py`, with input on HDFS and output written to HDFS. In Java, one would write a `Mapper` class with a `map()` method and a `Reducer` class with a `reduce()` method.

- **Result interpretation:** The output (word, count) pairs show the frequency of each word in the log. For example, `("error", 13)` means the word "error" appeared 13 times.

Assignment 12: Hadoop Weather Data

- **Mapper function:** The mapper reads each line of the weather data (e.g., CSV records with fields like date, location, temperature). It parses the line and emits key-value pairs relevant to the analysis. For example, to find yearly max temperature, the mapper might extract the year and temperature, then output `(year, temperature)`.

- **Reducer function:** The reducer receives all values for each key (year). It can compute an aggregate such as average or maximum temperature for that year. For instance, it might sum all temperatures and divide by count for the year, or keep track of the highest value. It then outputs `(year, result)`.

- **Aggregation example:** If the task is average temperature per year, reducer code might do `sum_temp = 0; count = 0; for temp in values: sum_temp += temp; count += 1; emit (year, sum_temp/count)`.

- **Hadoop streaming:** As with word count, one can use streaming to run mapper/reducer scripts. The final output shows a key (like year or station ID) and a computed weather statistic.

- **Purpose:** This MapReduce program processes a large weather dataset in parallel, demonstrating how to summarize or analyze climate data across many records.