Data preprocessing:

Certainly! Here’s a Python cheat sheet for the essential functions used in data preprocessing for machine learning, including the libraries needed and examples for reading and storing a .csv file. I’ll cover each of the topics you mentioned.

Libraries Required

```
import pandas as pd

import numpy as np

import seaborn as sns

import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler, MinMaxScaler, OneHotEncoder,
LabelEncoder
```

1. Handling Missing Data

pythonCopy code

```
# Load data

df = pd.read_csv('data.csv')

# Check missing values

df.isnull().sum()

# Drop rows with missing values

df.dropna(inplace=True)

# Fill missing values with mean of the column

df.fillna(df.mean(), inplace=True)


# Numerical columns: impute with mean

numerical_cols = df.select_dtypes(include=['int64', 'float64']).columns
```

```
# Categorical columns: impute with most frequent
categorical_cols = df.select_dtypes(include=['object']).columns

# Impute missing values
for col in numerical_cols:
    df[col].fillna(df[col].mean(), inplace=True)
```

2. Formatting Date Columns

```
# Convert a column to datetime
df['date_column'] = pd.to_datetime(df['date_column'])
# Extract year, month, day, etc.
df['year'] = df['date_column'].dt.year
df['month'] = df['date_column'].dt.month
df['day'] = df['date_column'].dt.day
```

3. Visualising Dataset Before and After Handling Missing Values

```
# Before
sns.heatmap(df.isnull(), cbar=False)
plt.show()
# After handling missing data
df.fillna(df.mean(), inplace=True)
sns.heatmap(df.isnull(), cbar=False)
plt.show()
```

4. Handling Outliers Using IQR

```
# Calculate IQR
Q1 = df['column'].quantile(0.25)
```

Q3 = df['column'].quantile(0.75)

IQR = Q3 - Q1

# Filter out outliers

df = df[(df['column'] >= (Q1 - 1.5 * IQR)) & (df['column'] <= (Q3 + 1.5 * IQR))]

for multiple columns:

```
for col in numerical_cols:
    if col != 'type':  # 'type' is now encoded, no longer numerical
        Q1 = df[col].quantile(0.25)
        Q3 = df[col].quantile(0.75)
        IQR = Q3 - Q1
        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR
        df[col] = df[col].clip(lower=lower_bound, upper=upper_bound)
```

5. Visualising Outliers Before and After Preprocessing

# Before

sns.boxplot(x=df['column'])

plt.show()

# After removing outliers

filtered_df = df[(df['column'] >= (Q1 - 1.5 * IQR)) & (df['column'] <= (Q3 + 1.5 * IQR))]

sns.boxplot(x=filtered_df['column'])

plt.show()

6. Categorical Encoding - One Hot and Label

# One Hot Encoding

df_encoded = pd.get_dummies(df, columns=['categorical_column'])

# Label Encoding

label_encoder = LabelEncoder()

df['categorical_column'] = label_encoder.fit_transform(df['categorical_column'])

7. Feature Scaling - Standardisation, Normalisation

# Standardisation

scaler = StandardScaler()

df['scaled_column'] = scaler.fit_transform(df[['numeric_column']])

# Normalisation

min_max_scaler = MinMaxScaler()

df['normalized_column'] = min_max_scaler.fit_transform(df[['numeric_column']])

8. Splitting the Dataset into Test and Train

# Split dataset

X_train, X_test, y_train, y_test = train_test_split(df.drop('target_column', axis=1), df['target_column'], test_size=0.2, random_state=42)

Reading and Storing a .csv File

# Reading a CSV file

df = pd.read_csv('data.csv')

# Storing a DataFrame as a CSV file

df.to_csv('processed_data.csv', index=False)

Regression:

Libraries Required

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import statsmodels.api as sm
from sklearn.linear_model import LinearRegression
```

1. Produce Scatterplots for Each Independent Variable Against the Dependent Variable (Price)

```
# Assuming 'price' is the dependent variable and others are independent variables
independent_vars = df.columns.drop('price')  # Adjust based on your dataset
for var in independent_vars:
    sns.scatterplot(x=df[var], y=df['price'])
    plt.title(f'Scatterplot of {var} vs Price')
    plt.xlabel(var)
    plt.ylabel('Price')
    plt.show()
```

2. Interpret the Strength of the Relationship Via the Correlation Matrix

```
# Calculate correlation matrix
correlation_matrix = df.corr()

# Plot heatmap of correlation matrix
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm')
plt.title('Correlation Matrix')
plt.show()
```

3. Fit a Linear Model to the Data to Predict the Prices of Avocados Using Linear Regression

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

from sklearn.metrics import mean_squared_error, r2_score

y_pred_train = lin_reg.predict(X_train)
y_pred_test = lin_reg.predict(X_test)

print('Train R-squared:', r2_score(y_train, y_pred_train))
print('Test R-squared:', r2_score(y_test, y_pred_test))
print('Train MSE:', mean_squared_error(y_train, y_pred_train))
print('Test MSE:', mean_squared_error(y_test, y_pred_test))
```

4. Assess the Strength of the Relationship Through statsmodels.summary()

```
import statsmodels.api as sm

X_train_sm = sm.add_constant(X_train)
model = sm.OLS(y_train, X_train_sm).fit()
print(model.summary())
```

5. Based on the Scatterplots and Correlation Values, Identify Which Variable(s) You Will Use to Predict the Price of Avocados

```
# Method 2: Correlation Matrix Analysis
correlation_matrix = train_df.corr()
correlation_with_target =
correlation_matrix['AveragePrice'].sort_values(ascending=False)
correlation_with_target = correlation_with_target.drop('AveragePrice')
print("Correlation matrix:\n", correlation_with_target)

# Determine thresholds based on percentiles
# For positive correlations
positive_threshold = correlation_with_target[correlation_with_target >
0].quantile(0.75)
# For negative correlations
negative_threshold = correlation_with_target[correlation_with_target <
0].quantile(0.25)

# Select variables that are highly positively and negatively correlated
based on these thresholds
highly_positively_correlated_vars =
correlation_with_target[correlation_with_target >
positive_threshold].index.tolist()
highly_negatively_correlated_vars =
correlation_with_target[correlation_with_target <
negative_threshold].index.tolist()

# Output the selected variables
print("Highly positively correlated variables:",
highly_positively_correlated_vars)
print("Highly negatively correlated variables:",
highly_negatively_correlated_vars)




import statsmodels.api as sm
```

```
# Assuming X_train and y_train are already defined and preprocessed
X_train_sm = sm.add_constant(X_train)  # Adding a constant for the
intercept
model = sm.OLS(y_train, X_train_sm).fit()

# Select significant variables based on p-value < 0.05
significant_vars = list(model.pvalues[model.pvalues < 0.01].index)
if 'const' in significant_vars:
    significant_vars.remove('const')  # Remove the constant

# Print the names of the significant variables and their p-values
print("Significant variables and their p-values:")
for var in significant_vars:
    print(f"{var}: p-value={model.pvalues[var]}")



# Combine lists from correlation analysis and OLS summary
final_vars = list(set(highly_positively_correlated_vars +
highly_negatively_correlated_vars + significant_vars ))

print(final_vars)
```

6. Use the Appropriate Independent Variables and Fit a Linear Model

```
# Fit a linear model using only the selected variables
X_train_final = X_train[final_vars]
X_test_final = X_test[final_vars]

lin_reg_final = LinearRegression()
lin_reg_final.fit(X_train_final, y_train)

# Predict and evaluate the final model using test data
y_pred_final = lin_reg_final.predict(X_test_final)

# Calculate metrics for the new model
mse_final = mean_squared_error(y_test, y_pred_final)
r2_final = r2_score(y_test, y_pred_final)

print(f'New Final model MSE: {mse_final}')
print(f'New Final model R-squared: {r2_final}')
```

Visualize the Results

```
# Create a scatter plot of actual vs. predicted values
plt.figure(figsize=(10, 6))
plt.scatter(y_test, y_pred_final, alpha=0.3)

# Plot a line of perfect prediction
min_val = min(y_test.min(), y_pred_final.min())
max_val = max(y_test.max(), y_pred_final.max())
plt.plot([min_val, max_val], [min_val, max_val], color='red',
linestyle='--', lw=2)

plt.title('Actual vs. Predicted Prices')
plt.xlabel('Actual Prices')
plt.ylabel('Predicted Prices')
plt.show()
```

LOGISTIC REGRESSION:

Fit model and confusion matrix

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix, classification_report

# Creating a pipeline that first scales the data then applies logistic
regression
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('logistic', LogisticRegression(solver='liblinear', max_iter=1000))
])

# Fitting the model using the pipeline. The pipeline will first transform
the data using StandardScaler and then fit the logistic regression model.
pipeline.fit(X_train, y_train)

# You can access the logistic regression model directly via
pipeline.named_steps['logistic']
logistic_model = pipeline.named_steps['logistic']


# The model is now fitted and can be used to make predictions on the
scaled test data automatically
y_pred = pipeline.predict(X_test)
```

```
# Generating the confusion matrix
cm = confusion_matrix(y_test, y_pred)

# Visualizing the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=['Predicted
0', 'Predicted 1'], yticklabels=['Actual 0', 'Actual 1'])
plt.title('Confusion Matrix')
plt.ylabel('Actual label')
plt.xlabel('Predicted label')
plt.show()
```

precision, recall and f1

```
# Classification Report for precision, recall, and F1 score
print(classification_report(y_test, y_pred, target_names=['No Rain',
'Rain']))

# Calculating and Displaying Specificity and Classification Error
tn, fp, fn, tp = cm.ravel()
specificity = tn / (tn + fp)
classification_error = (fp + fn) / (tp + tn + fp + fn)

print(f"Specificity: {specificity:.2f}")
print(f"Classification Error: {classification_error:.2f}")
```

ROC curve:

```
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt

# Assuming `pipeline` is your fitted model and it can predict
probabilities
# Get the scores (probabilities) of the positive class
y_scores = pipeline.predict_proba(X_test)[:, 1]

# Generate ROC curve values: false positive rates, true positive rates
fpr, tpr, thresholds = roc_curve(y_test, y_scores)

# Calculate Area Under the Curve (AUC)
roc_auc = auc(fpr, tpr)

# Plotting
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area =
%0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC)')
plt.legend(loc="lower right")
plt.show()
```

K fold cross validation

```
from sklearn.model_selection import cross_val_score, StratifiedKFold

# Initialize Stratified K-Fold to maintain the percentage of samples for
each class
kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Perform k-fold cross-validation
# Here we use 'accuracy' as the score to evaluate. You can choose other
metrics like 'roc_auc', 'f1', etc.
scores = cross_val_score(pipeline, X, y, cv=kfold, scoring='accuracy')

# Print the accuracy for each fold
print("Accuracy for each fold: ", scores)

# Print the mean accuracy across all folds
print("Mean cross-validation accuracy: ", scores.mean())


# Compare this mean accuracy to your baseline model's accuracy to check
for improvement
from sklearn.dummy import DummyClassifier

# Assuming y is your target variable from the dataframe 'df'
dummy = DummyClassifier(strategy='most_frequent', random_state=42)
dummy_scores = cross_val_score(dummy, X, y, cv=kfold, scoring='accuracy')

# Print the mean accuracy for the baseline model
print("Mean baseline accuracy: ", dummy_scores.mean())

# Now you can compare it to your logistic regression model's accuracy
print("Mean logistic regression accuracy: ", scores.mean())
print("Improvement over baseline: ", scores.mean() - dummy_scores.mean())
```

NAÏVE BAYES:

```
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
```

Fitting the naive bayes model

```
# Creating a pipeline that includes scaling and Gaussian Naive Bayes
pipeline = make_pipeline(StandardScaler(), GaussianNB())

# Fitting the model to the training data
pipeline.fit(X_train, y_train)

# Making predictions on the test set
y_pred = pipeline.predict(X_test)

# Evaluating the model
accuracy = accuracy_score(y_test, y_pred)
print(f"Test Accuracy: {accuracy}")
```

confusion matrix

```
from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns
import matplotlib.pyplot as plt

# Assuming y_test are the true labels and y_pred are the predictions made
by the model

# Generating the confusion matrix
cm = confusion_matrix(y_test, y_pred)

# Visualizing the confusion matrix
plt.figure(figsize=(8, 6))
```

```
sns.heatmap(cm, annot=True, fmt="d", cmap='Blues', xticklabels=['Predicted
No', 'Predicted Yes'], yticklabels=['Actual No', 'Actual Yes'])
plt.ylabel('True label')
plt.xlabel('Predicted label')
plt.title('Confusion Matrix for Naive Bayes Model')
plt.show()
```

recall, f1, precision

```
from sklearn.metrics import classification_report

# Generate and print the classification report for precision, recall, and
F1 score
print(classification_report(y_test, y_pred, target_names=['No', 'Yes']))

# Calculate the confusion matrix to use for calculating classification
error and specificity
cm = confusion_matrix(y_test, y_pred)
tn, fp, fn, tp = cm.ravel()

# Classification error (also known as Misclassification Rate)
classification_error = (fp + fn) / float(tp + tn + fp + fn)
print(f"Classification Error: {classification_error:.2f}")

# Specificity (True Negative Rate)
specificity = tn / (tn + fp)
print(f"Specificity: {specificity:.2f}")
```

ROC curve

```
from sklearn.metrics import roc_curve, roc_auc_score
import numpy as np
import matplotlib.pyplot as plt


# Step 1: Get the predicted probabilities for the positive class
y_scores = pipeline.predict_proba(X_test)[:, 1]

# Step 2: Calculate ROC curve and AUC score
fpr, tpr, thresholds = roc_curve(y_test, y_scores)
roc_auc = roc_auc_score(y_test, y_scores)

# Step 3: Plot ROC curve
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, label=f'ROC curve (AUC = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], 'k--')  # Dashed diagonal
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend(loc='lower right')
plt.show()
```

adjusting the decision threshold based on Youden's J statistic

```
from sklearn.metrics import roc_curve
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score

# Assume y_scores are the predicted probabilities for the positive class
fpr, tpr, thresholds = roc_curve(y_test, y_scores)

# Calculate Youden's J statistic
j_statistic = tpr - fpr
optimal_idx = np.argmax(j_statistic)
optimal_threshold = thresholds[optimal_idx]

print(f"Optimal Threshold based on Youden's J statistic:
{optimal_threshold}")

# Step 4: Choose a new threshold from the ROC curve that suits your need
# This is a manual step depending on your specific requirement (e.g.,
balancing precision and recall)
# For demonstration, let's say you chose a threshold that gives you higher
recall
new_threshold = optimal_threshold  # Example threshold

# Step 5: Apply the new threshold to adjust classification decisions
y_pred_adjusted = (y_scores >= new_threshold).astype(int)

# Now you can calculate metrics using the adjusted predictions to see the
impact
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score

print("Adjusted Metrics:")
print(f"Accuracy: {accuracy_score(y_test, y_pred_adjusted):.4f}")
print(f"Precision: {precision_score(y_test, y_pred_adjusted):.4f}")
print(f"Recall: {recall_score(y_test, y_pred_adjusted):.4f}")
print(f"F1 Score: {f1_score(y_test, y_pred_adjusted):.4f}")
```

k-fold cross validation

```
from sklearn.model_selection import cross_val_score, StratifiedKFold
from sklearn.naive_bayes import GaussianNB
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

# Assuming 'X' and 'y' are your features and target variable from the
preprocessed DataFrame

# Creating a pipeline with a scaler and Gaussian Naive Bayes
pipeline = make_pipeline(StandardScaler(), GaussianNB())

# Evaluating model performance with cross-validation
kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
cv_scores = cross_val_score(pipeline, X, y, cv=kfold, scoring='accuracy')

print(f"CV Accuracy: {cv_scores.mean()} ± {cv_scores.std()}")
```

tuning hyperparameters

```
from sklearn.model_selection import GridSearchCV
from sklearn.naive_bayes import GaussianNB
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

# Assuming 'X' and 'y' are your features and target variable from the
preprocessed DataFrame

# Creating a pipeline that includes scaling and Gaussian Naive Bayes
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('naive_bayes', GaussianNB())
])

# Define a grid of hyperparameters to search
# For GaussianNB, we can tune the 'var_smoothing' parameter
param_grid = {
    'naive_bayes__var_smoothing': np.logspace(0, -9, num=100)
}

# Set up GridSearchCV
grid_search = GridSearchCV(pipeline, param_grid, cv=5, scoring='accuracy',
verbose=1)

# Fit the GridSearchCV object to the data
```

```
grid_search.fit(X, y)

# Print the best parameters and the best score
print("Best parameters found: ", grid_search.best_params_)
print("Best cross-validation accuracy: ", grid_search.best_score_)
```

Clustering:

K-means clustering

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.cluster import KMeans
```

Elbow method to find number of clusters

```
# Assuming you've saved the cleaned data to 'cleaned_data.csv'
data_scaled =
pd.read_csv('/content/drive/MyDrive/dataset/Live_scaled.csv')

# Select features for clustering
X = data_scaled[['num_reactions', 'num_comments']].values

# Apply the Elbow Method to find the optimal number of clusters
wcss = []
for i in range(1, 11):
    kmeans = KMeans(n_clusters=i, init='k-means++', n_init=10,
random_state=42)
    kmeans.fit(X)
    wcss.append(kmeans.inertia_)

plt.plot(range(1, 11), wcss)
plt.title('The Elbow Method')
plt.xlabel('Number of clusters')
plt.ylabel('WCSS')
plt.show()
```

number of cluster fining using knee locator (optional)

```
from kneed import KneeLocator
```

```
import matplotlib.pyplot as plt

# Number of clusters range
range_n_clusters = range(1, 11)

# Find the elbow point
knee_locator = KneeLocator(range_n_clusters, wcss, curve='convex',
direction='decreasing')

# Optimal number of clusters
optimal_clusters = knee_locator.elbow

print(f"The optimal number of clusters: {optimal_clusters}")
```

k-means cluster fitting

```
# Perform K-means clustering
kmeans = KMeans(n_clusters=3, init='k-means++', n_init=10,
random_state=42)
y_kmeans = kmeans.fit_predict(X)

# Visualize the clusters
plt.scatter(X[y_kmeans == 0, 0], X[y_kmeans == 0, 1], s=100, c='red',
label='Cluster 1')
plt.scatter(X[y_kmeans == 1, 0], X[y_kmeans == 1, 1], s=100, c='blue',
label='Cluster 2')
plt.scatter(X[y_kmeans == 2, 0], X[y_kmeans == 2, 1], s=100, c='green',
label='Cluster 3')

# Plotting the centroids of the clusters
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
s=300, c='yellow', label='Centroids')
plt.title('K-means Clusters of Facebook Live Sellers')
plt.xlabel('Number of Reactions')
plt.ylabel('Number of Comments')
plt.legend()
plt.show()
```

Print major element in each cluster:

```
# Add cluster labels to the original cleaned data (not scaled)
data_cleaned['Cluster'] = y_kmeans

# Ensure this uses the original, unscaled 'data_cleaned' with 'Cluster'
labels assigned
for cluster in sorted(data_cleaned['Cluster'].unique()):
    cluster_data = data_cleaned[data_cleaned['Cluster'] == cluster]
    # Find the encoded majority status_type, ensuring to work with
non-scaled, encoded values
    majority_status_encoded = cluster_data['status_type'].mode()[0]

    # Decode the majority status_type
    majority_status =
label_encoder.inverse_transform([int(majority_status_encoded)])[0]

    print(f"Majority status_type in Cluster {cluster}: {majority_status}")
```

HIERARCHICAL CLUSTERING:

dendrogram to find the optimal number of clusters

```
import scipy.cluster.hierarchy as sch
import matplotlib.pyplot as plt

# Create a dendrogram
dendrogram = sch.dendrogram(sch.linkage(X, method='ward'))
plt.title('Dendrogram')
plt.xlabel('Data Points')
plt.ylabel('Euclidean Distances')
plt.show()
```

Hierarchical Clustering model fitting

```
from sklearn.cluster import AgglomerativeClustering

# Perform hierarchical clustering
hc = AgglomerativeClustering(n_clusters=3, affinity='euclidean',
linkage='ward')
y_hc = hc.fit_predict(X)
```

Visualizing the clusters

```
# Visualize the clusters
plt.scatter(X[y_hc == 0, 0], X[y_hc == 0, 1], s=100, c='red',
label='Cluster 1')
plt.scatter(X[y_hc == 1, 0], X[y_hc == 1, 1], s=100, c='blue',
label='Cluster 2')
plt.scatter(X[y_hc == 2, 0], X[y_hc == 2, 1], s=100, c='green',
label='Cluster 3')
```

```
plt.title('Clusters of Facebook Live Sellers (Hierarchical Clustering)')
plt.xlabel('Number of Reactions')
plt.ylabel('Number of Comments')
plt.legend()
plt.show()
```

Majority status type in each cluster

```
# Add the cluster labels to your original dataset
data['Cluster'] = y_hc

# Label encode 'status_type' to revert back to original categorical types
for majority voting
label_encoder = LabelEncoder()
data['status_type_encoded'] =
label_encoder.fit_transform(data['status_type'])

# Calculate the majority 'status_type' for each cluster
for i in range(3):  # assuming 3 clusters
    cluster_group = data[data['Cluster'] == i]
    majority_status = cluster_group['status_type'].value_counts().idxmax()
    print(f"Majority status_type in Cluster {i}: {majority_status}")
```

Decision Tree:

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

dt = DecisionTreeClassifier(random_state=42)
dt.fit(X_train, y_train)

y_pred_dt = dt.predict(X_test)
print(f"Decision Tree Accuracy: {accuracy_score(y_test, y_pred_dt)}")
```

hyperparameter tuning

```
from sklearn.model_selection import GridSearchCV

dt_param_grid = {
    'max_depth': [None, 10, 20, 30],
    'min_samples_leaf': [1, 2, 4],
    'min_samples_split': [2, 5, 10]
}

dt_grid_search = GridSearchCV(DecisionTreeClassifier(random_state=42),
dt_param_grid, cv=5, scoring='accuracy')
dt_grid_search.fit(X_train, y_train)

print(f"Best parameters for Decision Tree: {dt_grid_search.best_params_}")
```

```
# Using the best estimator from the grid search
best_dt = dt_grid_search.best_estimator_
y_pred_best_dt = best_dt.predict(X_test)
print(f"Decision Tree Accuracy after tuning: {accuracy_score(y_test,
y_pred_best_dt)}")
```

Neural networks:

```
from sklearn.neural_network import MLPClassifier

nn = MLPClassifier(random_state=42, max_iter=1000)
nn.fit(X_train, y_train)  # Ensure X_train is scaled

y_pred_nn = nn.predict(X_test)
print(f"Neural Network Accuracy: {accuracy_score(y_test, y_pred_nn)}")
```

hyperparameter tuning

```
nn_param_grid = {
    'hidden_layer_sizes': [(50,), (100,), (50,50), (100,100)],
    'activation': ['tanh', 'relu'],
    'solver': ['sgd', 'adam'],
    'alpha': [0.0001, 0.05],
}

nn_grid_search = GridSearchCV(MLPClassifier(random_state=42,
max_iter=1000), nn_param_grid, cv=5, scoring='accuracy')
nn_grid_search.fit(X_train, y_train)

print(f"Best parameters for Neural Network:
{nn_grid_search.best_params_}")
```

```
best_nn = nn_grid_search.best_estimator_
y_pred_best_nn = best_nn.predict(X_test)
print(f"Neural Network Accuracy after tuning: {accuracy_score(y_test,
y_pred_best_nn)}")
```

Data Preparation:

https://medium.com/@vaniksaras/mastering-the-art-of-data-alchemy-a-guide-to-seamless-data-preparation-11b9191dbce3

What is Regression?

Regression is a statistical method used in finance, investing, and other disciplines that attempts to determine the strength and character of the relationship between one dependent variable (usually denoted by Y) and a series of other changing variables (known as independent variables). The main goal of regression is to predict or explain the dependent variable based on the independent variables. It is widely used for prediction and forecasting in many fields.

How to Perform Regression Analysis

Performing regression typically involves the following steps:

Data Collection: Gather the data that will be used in the prediction model.

Exploratory Data Analysis: Check how the data is distributed and identify any correlations among the data points.

Data Cleaning and Preparation: Handle missing data, outliers, and normalize the data if necessary.

Selecting the Model: Choose the type of regression model that best fits the problem.

Model Training: Train the model using a part of the dataset (training dataset).

Model Evaluation: Evaluate the model's performance using a different part of the dataset (testing dataset). Common metrics include R-squared, Root Mean Squared Error (RMSE), and Mean Absolute Error (MAE).

Model Refinement: Fine-tune the model by adjusting parameters or model configuration based on performance metrics.

Prediction: Use the model to make predictions.

Types of Regression and Their Uses

There are several types of regression models, each suited for different types of data and analysis needs:

Linear Regression

Description: Predicts a dependent variable value (y) based on a given independent variable (x). If this relationship is plotted on a 2D space, the data points should ideally form a straight line.

Use Case: Used when data points form a linearly distributable dataset, such as predicting house prices based on area or predicting salary based on experience.

Multiple Linear Regression

Description: Similar to linear regression, but with multiple independent variables contributing to the dependent variable's outcome.

Use Case: Useful for more complex problems where multiple variables affect the dependent variable, like predicting a car's mileage based on engine size, weight, and horsepower.

Polynomial Regression

Description: Extends linear regression by adding terms with powers greater than one, making it suitable for non-linear data sets.

Use Case: Used when the data points form a curve rather than a straight line, such as the growth rate of tissues or the progression of disease epidemics.

Logistic Regression

Description: Despite its name, it’s a linear model for classification rather than regression. It predicts a probability that Y belongs to a certain category.

Use Case: Commonly used for binary classification tasks, like spam detection or predicting if a student will pass or fail based on their hours of study.

Ridge Regression

Description: A type of regularized linear regression that includes a penalty term to reduce model complexity and prevent overfitting.

Use Case: Useful when the data includes multicollinearity or when you have more features than data points.

Lasso Regression

Description: Similar to Ridge Regression but with a different penalty term that can reduce the number of features in the final model by assigning zero coefficients to less important features.

Use Case: Ideal for models that benefit from feature selection in case of high dimensionality.

Elastic Net Regression

Description: Combines penalties of Ridge and Lasso regression to provide a balanced approach to regularizing complex models.

Use Case: Used when there are several highly correlated variables, effectively balancing between parameter penalty and model complexity.

Linear classification:

Linear classification involves predicting a categorical outcome based on one or more input features. The prediction is made by calculating a linear combination of the input features and applying a decision rule based on the output of the linear model. Here are some important concepts and functions associated with linear classification:

1. Concepts of Linear Classification

a. Decision Boundary

- **Definition**: In linear classification, the decision boundary is a hyperplane that separates the classes. The decision is based on which side of the boundary an input point falls on.
- **Example**: In a 2D space, the decision boundary is a line. If the classifier is a linear regression model, this boundary could be represented as $y = wx + b$, where w is the weight, x is the input feature, and b is the bias.

b. Margin

- **Definition**: The margin in a linear classifier refers to the distance between the decision boundary and the closest data points from any class. Maximizing this margin can help in reducing generalization errors.
- **Relevance**: This concept is crucial in Support Vector Machines (SVM), where the goal is to find

c. Loss Functions

- **Hinge Loss**: Used primarily with SVM. It is defined as $\max(0, 1 - y_i \cdot (w \cdot x_i + b))$, where y_i is the true label, $w \cdot x_i + b$ is the predicted value, and y_i can be +1 or -1.
- **Logistic Loss**: Used for logistic regression, represented as $\log(1 + e^{-y_i(w \cdot x_i + b)})$.

d. Regularization

- **Purpose**: To prevent overfitting by penalizing large coefficients in the model.
- **L1 Regularization (Lasso)**: Adds a penalty equal to the absolute value of the magnitude of coefficients.
- **L2 Regularization (Ridge)**: Adds a penalty equal to the square of the magnitude of coefficients.

2. Important Functions for Linear Classification in Python (`sklearn`)

a. Logistic Regression (`sklearn.linear_model.LogisticRegression`)

python Copy code

```python
from sklearn.linear_model import LogisticRegression

model = LogisticRegression()
model.fit(X_train, y_train)
predictions = model.predict(X_test)
```

3. Model Evaluation Metrics

- **Accuracy**: Fraction of predictions our model got right.
- **Precision and Recall**: Precision is the accuracy of positive predictions. Recall (also known as sensitivity) measures the ability of a classifier to find all the positive samples.
- **F1 Score**: Harmonic mean of precision and recall.

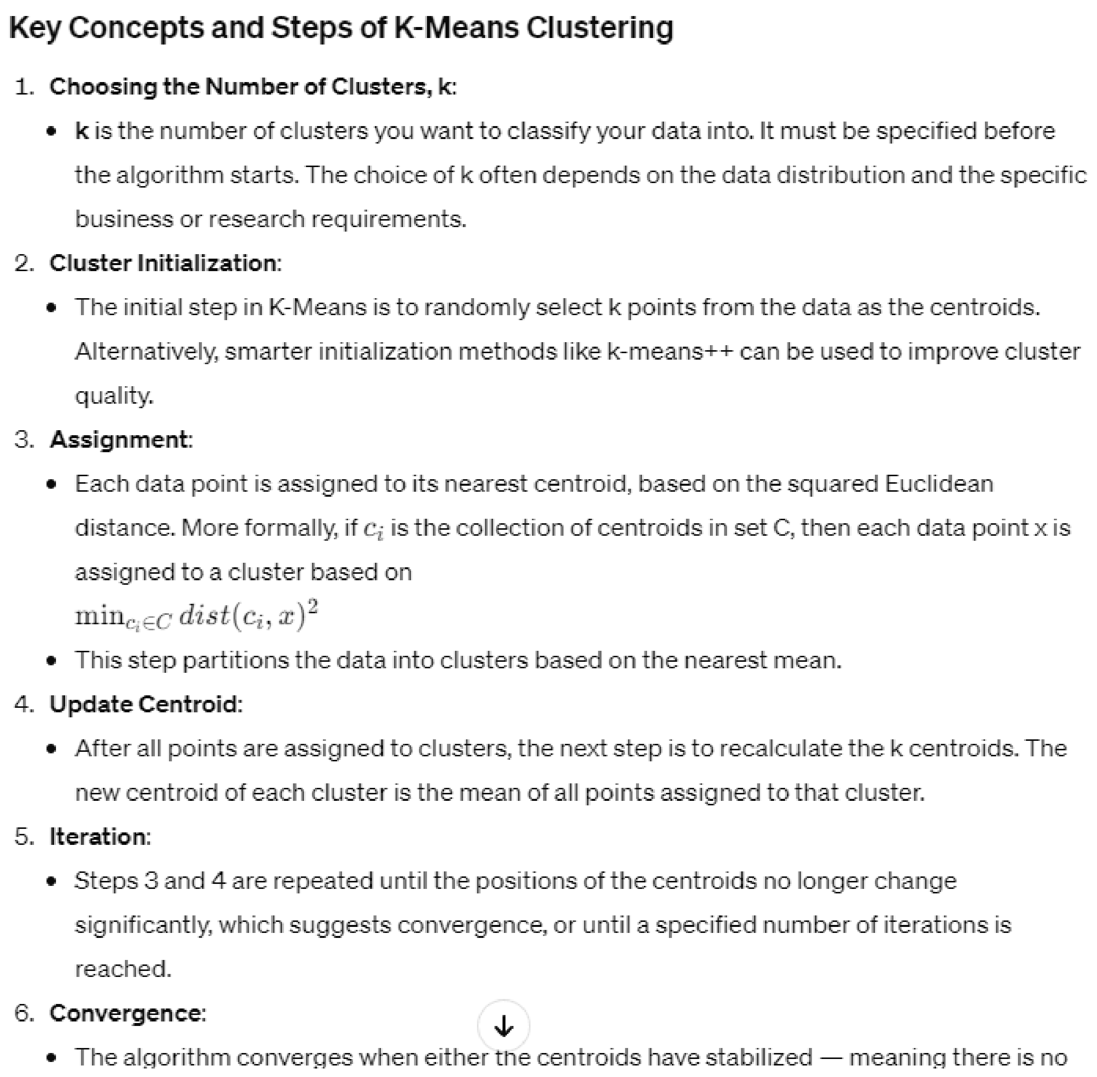
4. Feature Scaling

- **Purpose**: Ensures that all features contribute equally to the result and helps gradient descent converge faster.
- **Methods**: Min-Max scaling, Standardization (Z-score normalization).

5. Interpretation of Model Coefficients

- **Coefficients**: In linear models, each coefficient tells the change in the log odds for a one unit increase in the corresponding feature, holding all other features constant.

K means clustering



Key Concepts and Steps of K-Means Clustering

1. **Choosing the Number of Clusters, k**:
   - **k** is the number of clusters you want to classify your data into. It must be specified before the algorithm starts. The choice of k often depends on the data distribution and the specific business or research requirements.
2. **Cluster Initialization**:
   - The initial step in K-Means is to randomly select k points from the data as the centroids. Alternatively, smarter initialization methods like k-means++ can be used to improve cluster quality.
3. **Assignment**:
   - Each data point is assigned to its nearest centroid, based on the squared Euclidean distance. More formally, if c_i is the collection of centroids in set C, then each data point x is assigned to a cluster based on
     $\min_{c_i \in C} dist(c_i, x)^2$
   - This step partitions the data into clusters based on the nearest mean.
4. **Update Centroid**:
   - After all points are assigned to clusters, the next step is to recalculate the k centroids. The new centroid of each cluster is the mean of all points assigned to that cluster.
5. **Iteration**:
   - Steps 3 and 4 are repeated until the positions of the centroids no longer change significantly, which suggests convergence, or until a specified number of iterations is reached.
6. **Convergence**:
   - The algorithm converges when either the centroids have stabilized — meaning there is no

ChatGPT

The Elbow Method is a heuristic used in clustering analysis, particularly with K-means clustering, to determine the optimal number of clusters (k). The method consists of plotting the explained variation as a function of the number of clusters, and picking the elbow of the curve as the number of clusters to use. This "elbow" point is considered to be where the within-cluster sum of squares (WCSS) starts decreasing at a slower rate, indicating that adding more clusters does not significantly improve the fitting of the data.

Steps to Implement the Elbow Method

Here's how you can implement the Elbow Method:

1. **Compute Clustering for Different Values of k**:
   - For a range of k values (e.g., from 1 to 10), perform the K-means clustering. For each k, you initialize the K-means algorithm, partition the dataset into k clusters, and compute the cluster centroids.
2. **Calculate the Within-Cluster Sum of Squares (WCSS)**:
   - For each cluster configuration, calculate the total sum of squared distances between data points and their respective cluster centroids. In mathematical terms, if c_i is the centroid of cluster C_i, then WCSS is calculated as:

$$\text{WCSS} = \sum_{i=1}^{k} \sum_{x \in C_i} (x - c_i)^2$$

   - The goal is to minimize WCSS.

3. **Plot the Results**:
   - Plot a line graph of the WCSS for each k value. The x-axis will represent the number of clusters, and the y-axis will represent the WCSS.
4. **Determine the Elbow Point**:
   - The location of a bend (elbow) in the plot is generally considered as an indicator of the appropriate number of clusters. This point represents where the WCSS starts to decrease linearly, suggesting that the addition of more clusters does not significantly improve the fit of the model.

Hierarchical clustering is a type of clustering algorithm that builds a hierarchy of clusters where each node is a cluster consisting of the clusters of its daughter nodes. Strategies for hierarchical clustering generally fall into two types:

1. **Agglomerative**: This is a "bottom-up" approach starting with each data point as a single cluster and then pairs of clusters are successively merged as one moves up the hierarchy.
2. **Divisive**: This is a "top-down" approach starting with all data points in one cluster, then splits are performed recursively as one moves down the hierarchy.

Key Concepts of Hierarchical Clustering

- **Dendrogram**: The dendrogram is a tree-like diagram that records the sequences of merges or splits.

Steps in Agglomerative Hierarchical Clustering

1. **Initialization**: Start by assigning each data point to its own cluster, so if you have N data points, you initially have N clusters, each containing just one data point.
2. **Compute the Proximity Matrix**: Calculate the distance between each pair of clusters. Common distance metrics used are:
   - **Euclidean Distance**: Usual straight-line distance between two points in Euclidean space.
   - **Manhattan Distance**: Sum of the absolute differences of their Cartesian coordinates.
   - **Cosine Similarity**: Measures cosine of the angle between two non-zero vectors.
3. **Merge the Closest Pairs of Clusters**: Find the two clusters that are closest together and merge them into a single cluster. This decreas [illegible]e total number of clusters by one.
4. **Update the Proximity Matrix**: After merging two clusters, we need to update the distances

4. **Update the Proximity Matrix**: After merging two clusters, we need to update the distances between the new cluster and each of the old clusters.
5. **Repeat Steps 3 and 4**: Continue merging the closest pair of clusters and updating the distance matrix until all the data points are clustered into a single cluster of size N.
6. **Generate the Dendrogram**: A dendrogram is a diagram representing a tree. This diagrammatic representation is useful for understanding the data structure during different stages of clustering.

Types of Linkage Methods

When merging clusters, the choice of the linkage method, which determines the distance between sets of observations as a function of the pairwise distances between observations, can significantly affect the shape of the clusters:

- **Single Linkage**: Nearest distance. Distance between two clusters is defined as the shortest distance between points in the two clusters.
- **Complete Linkage**: Farthest distance. Distance between two clusters is defined as the longest distance between points in the two clusters.
- **Average Linkage**: Average distance. Distance between two clusters is defined as the average distance between each point in one cluster to every point in the other cluster.
- **Ward's Method**: Minimize variance. Merges the pair of clusters that leads to minimum increase in total within-cluster variance after merging.

A dendrogram is a tree-like diagram that represents the hierarchical relationships between various objects in a dataset. It's particularly associated with hierarchical clustering methods and effectively visualizes the arrangement of clusters formed at each stage of the analysis. Dendrograms are valuable tools in exploratory data analysis, helping identify the number and composition of clusters in data.

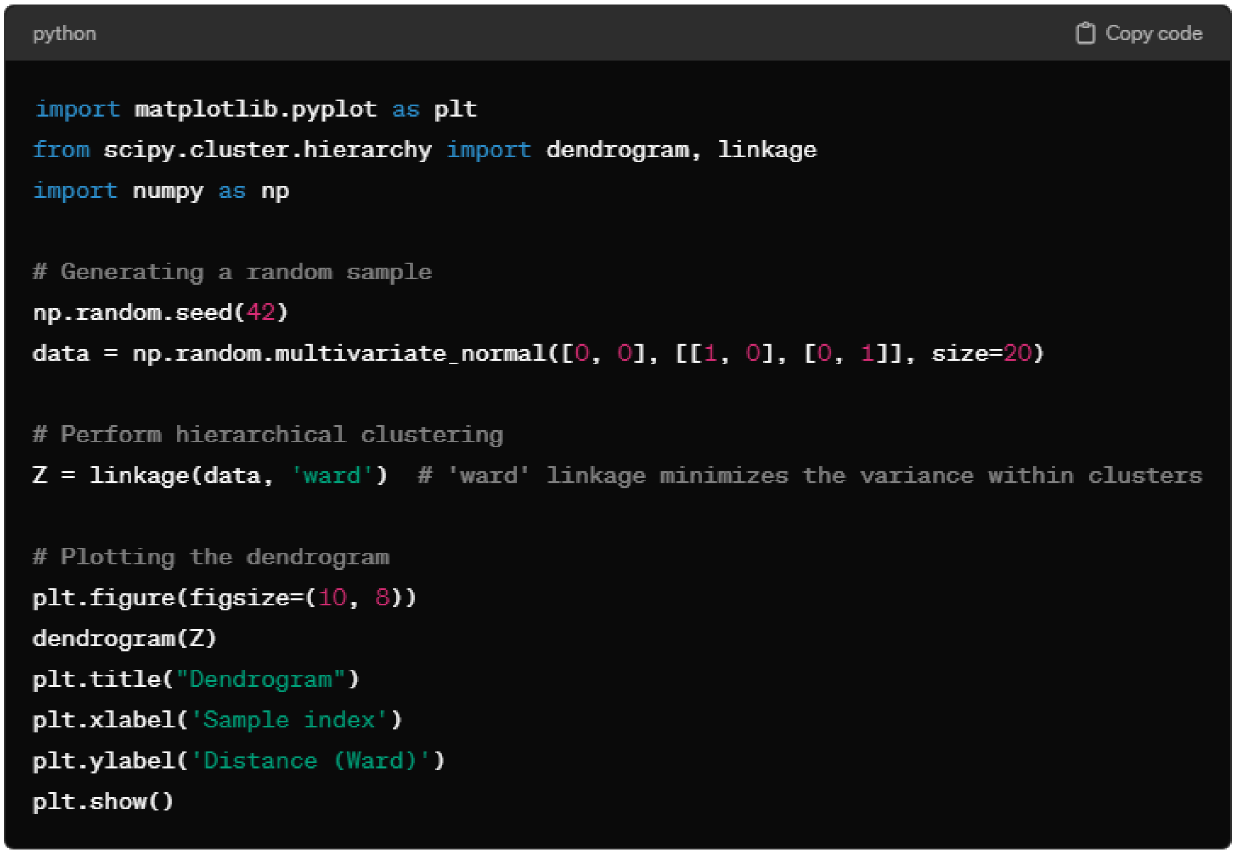
Structure of a Dendrogram

In a dendrogram:

- Each leaf represents an individual data point or object in the dataset.
- As you move from the leaves to the root, individual leaves or groups of leaves merge into branches, which then merge with other branches or leaves.
- The height at which any two branches merge represents the distance between these groups in the data space. This distance can be determined using various metrics like Euclidean distance, Manhattan distance, etc., and the interpretation of this distance depends on the linkage method used (single, complete, average, Ward's, etc.).

How to Interpret a Dendrogram

1. **Height of Merges**: The y-axis of a dendrogram often shows the level of similarity or distance at which clusters are merged. A higher merge point indicates less similarity between clusters, suggesting that they are distinctly different from each other.
2. **Determining the Number of Clusters**:
   - To choose the number of clusters fro dendrogram, you can use a horizontal cut-line. Where you place this line determines the number of clusters. The line should ideally cut the tallest



```python
import matplotlib.pyplot as plt
from scipy.cluster.hierarchy import dendrogram, linkage
import numpy as np

# Generating a random sample
np.random.seed(42)
data = np.random.multivariate_normal([0, 0], [[1, 0], [0, 1]], size=20)

# Perform hierarchical clustering
Z = linkage(data, 'ward')  # 'ward' linkage minimizes the variance within clusters

# Plotting the dendrogram
plt.figure(figsize=(10, 8))
dendrogram(Z)
plt.title("Dendrogram")
plt.xlabel('Sample index')
plt.ylabel('Distance (Ward)')
plt.show()
```