DATA SCIENCE BRAIN
@datasciencebrain

DATA CLEANING CHEAT SHEET for Machine Learning

Save for later reference

01 HANDLING MISSING VALUES

1. Remove Rows with Missing Values

```
df = df.dropna()
```

2. Drop Columns with a Certain Percentage of Missing Values

```
df = df.dropna(thresh=len(df) * 0.7, axis=1)
```

3. Impute Missing Values in Numeric Columns with Mean

```
from sklearn.impute import SimpleImputer
numeric_cols = df.select_dtypes(include=['float64', 'int64']).columns
imputer = SimpleImputer(strategy='mean')
df[numeric_cols] = imputer.fit_transform(df[numeric_cols])
```

4. Impute Missing Values in Categorical Columns with Most Frequent

```
categorical_cols = df.select_dtypes(include='object').columns
imputer = SimpleImputer(strategy='most_frequent')
df[categorical_cols] = imputer.fit_transform(df[categorical_cols])
```

5. Interpolation for Time Series Data

```
df = df.interpolate()
```

6. K-Nearest Neighbors (KNN) Imputation

```
from sklearn.impute import KNNImputer
knn_imputer = KNNImputer(n_neighbors=2)
df = knn_imputer.fit_transform(df)
```

02 HANDLING OUTLIERS

Z-Score Method: Identify outliers based on the Z-scores of numeric columns.

```
z_scores = (df[numeric_cols] - df[numeric_cols].mean()) / df[numeric_cols].std()
df = df[(z_scores < 3).all(axis=1)]
```

This example filters out rows where the Z-score in any numeric column is greater than 3.

IQR (Interquartile Range) Method: Identify and remove outliers using the IQR.

```
Q1 = df[numeric_cols].quantile(0.25)
Q3 = df[numeric_cols].quantile(0.75)
IQR = Q3 - Q1
df = df[~((df[numeric_cols] < (Q1 - 1.5 * IQR)) | (df[numeric_cols] > (Q3 + 1.5 * IQR))).any(axis=1)]
```

This example uses the IQR to filter out rows with values outside the 1.5 times IQR range.

Visual Inspection and Manual Removal: Examine visualizations like box plots or scatter plots to identify and manually remove outliers based on domain knowledge.

```
import seaborn as sns
sns.boxplot(x=df['numeric_column'])
```

03 HANDLING DUPLICATES

Remove Exact Duplicates: Identify and remove rows that have the exact same values across all columns.

```
df = df.drop_duplicates()
```

Remove Duplicates Based on Specific Columns: Remove rows where specific columns have duplicate values.

```
columns_to_check = ['col1', 'col2', 'col3']
df = df.drop_duplicates(subset=columns_to_check)
```

Counting and Identifying Duplicates: Check for the existence of duplicates and display their count.

```
duplicate_count = df.duplicated().sum()
duplicate_rows = df[df.duplicated()]
```

Handling Duplicates While Keeping the Last Occurrence: Keep the last occurrence of a duplicate row based on a specific column.

```
df = df.drop_duplicates(subset='column_to_check',
keep='last')
```

04 ENCODING CATEGORICAL VARIABLES

One-Hot Encoding: Convert categorical variables into binary vectors.

```
df = pd.get_dummies(df, columns=categorical_cols, drop_first=True)
```

This creates binary columns for each category, dropping the first to avoid multicollinearity.

Label Encoding: Convert categorical labels into numerical values.

```
from sklearn.preprocessing import LabelEncoder
label_encoder = LabelEncoder()
df['categorical_column'] =
label_encoder.fit_transform(df['categorical_column'])
```

Binary Encoding: Encode categorical features as binary numbers.

```
import category_encoders as ce
binary_encoder = ce.BinaryEncoder(cols=categorical_cols)
df = binary_encoder.fit_transform(df)
```

Ordinal Encoding: Encode categorical variables as ordinal integers.

```
ordinal_mapping = {'low': 1, 'medium': 2, 'high': 3}
df['ordinal_column'] = df['ordinal_column'].map(ordinal_mapping)
```

05 HANDLING INCONSISTENT DATA TYPES

Ensure Proper Data Types: Ensure that each column has the correct data type.

```
df = df.astype({'numeric_column': 'float64',
'categorical_column': 'category'})
```

Use the astype method to explicitly set the data type of each column.

Convert Data Types: Convert data types, for example, from string to numeric.

```
df['numeric_column'] =
pd.to_numeric(df['numeric_column'], errors='coerce')
```

Use pd.to_numeric to convert a column to numeric, handling errors as specified.

Convert Text to Lowercase or Uppercase: Standardize text data by converting to lowercase or uppercase.

```
df['text_column'] = df['text_column'].str.lower()
```

Use str.lower() or str.upper() to standardize text data.

06 FEATURE SCALING

Standard Scaling (Z-score normalization): Scale numerical features to have a mean of 0 and a standard deviation of 1.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
df['numeric_column'] =
scaler.fit_transform(df[['numeric_column']])
```

Fit a StandardScaler to your numeric columns and transform them.

Min-Max Scaling: Scale numerical features to a specific range, often between 0 and 1.

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
df['numeric_column'] =
scaler.fit_transform(df[['numeric_column']])
```

Robust Scaling: Scale features using the median and interquartile range, suitable for data with outliers.

```
from sklearn.preprocessing import RobustScaler
scaler = RobustScaler()
df['numeric_column'] =
scaler.fit_transform(df[['numeric_column']])
```

08 HANDLING IMBALANCED DATASETS

Oversampling: Increase the number of instances of the minority class.

```
from imblearn.over_sampling import RandomOverSampler
oversampler = RandomOverSampler(sampling_strategy='minority')
X_resampled, y_resampled = oversampler.fit_resample(X, y)
```

Undersampling: Decrease the number of instances of the majority class.

```
from imblearn.under_sampling import RandomUnderSampler
undersampler =
RandomUnderSampler(sampling_strategy='majority')
X_resampled, y_resampled = undersampler.fit_resample(X, y)
```

SMOTE (Synthetic Minority Over-sampling Technique): Generate synthetic samples for the minority class.

```
from imblearn.over_sampling import SMOTE
smote = SMOTE(sampling_strategy='minority')
X_resampled, y_resampled = smote.fit_resample(X, y)
```

Class Weights: Adjust class weights during model training to give more importance to the minority class.

```
from sklearn.ensemble import RandomForestClassifier
classifier = RandomForestClassifier(class_weight='balanced')
```

08 ADDITIONAL STEPS

1. Handling Time Series Data:

```
# Handling missing values with forward fill for time series data
df = df.ffill()
```

2. Feature Engineering:

```
# Creating a new feature by combining existing ones
df['new_feature'] = df['feature1'] * df['feature2']
```

3. Handling Skewed Data:

```
# Applying log transformation to the target variableimport numpy as np
df['target_variable'] = np.log1p(df['target_variable'])
```

4. Memory Optimization:

```
# Convert columns to more memory-efficient types
df['numeric_column'] = df['numeric_column'].astype('float32')
```

5. Cross-Validation:

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(model, X, y, cv=5)
```

6. Pipeline Construction:

```
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer

# Example pipeline with preprocessing steps and model
preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numeric_cols),
        ('cat', OneHotEncoder(), categorical_cols)
    ])

pipeline = Pipeline(steps=[('preprocessor', preprocessor),
                           ('model', RandomForestClassifier())])

# Fit the pipeline
pipeline.fit(X_train, y_train)
```