

SHETH L.U.J. & SIR M.V. COLLEGE

Vedant Patil | T101

Practical No. 3

Aim: Feature Scaling and Dummification

- Apply feature-scaling techniques like standardization and normalization to numerical features.
- Perform feature dummification to convert categorical variables into numerical representations.

Part 1: Handling Numerical Data

1. Import Libraries and Load Data

```
import pandas as pd
import numpy as np
from sklearn import preprocessing
from sklearn.preprocessing import Normalizer
from sklearn.cluster import KMeans
from sklearn.impute import SimpleImputer

# Load your dataset
df = pd.read_csv('winequality_red.csv')
print("Original Data Head:")
print(df.head())
```

Original Data Head:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	\
0	7.4	0.70	0.00	1.9	0.076	
1	7.8	0.88	0.00	2.6	0.098	
2	7.8	0.76	0.04	2.3	0.092	
3	11.2	0.28	0.56	1.9	0.075	
4	7.4	0.70	0.00	1.9	0.076	

	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	\
0	11.0	34.0	0.9978	3.51	0.56	
1	25.0	67.0	0.9968	3.20	0.68	
2	15.0	54.0	0.9970	3.26	0.65	
3	17.0	60.0	0.9980	3.16	0.58	
4	11.0	34.0	0.9978	3.51	0.56	

	alcohol	quality
0	9.4	5
1	9.8	5
2	9.8	5
3	9.8	6
4	9.4	5

2. Rescaling a Feature (MinMax Scaling)

```
# Example with 'alcohol' column
feature_alcohol = df[['alcohol']].values
minmax_scaler = preprocessing.MinMaxScaler(feature_range=(0,1))
scaled_alcohol = minmax_scaler.fit_transform(feature_alcohol)
print("Scaled Alcohol (First 5 values):")
print(scaled_alcohol[:5].flatten())
```

Scaled Alcohol (First 5 values):

```
[0.15384615 0.21538462 0.21538462 0.21538462 0.15384615]
```

3. Standardizing a Feature (Z-Score & Robust)

```
# Standardize the 'pH' column (or any numerical column in your dataset)
feature_ph = df[['pH']].values
scaler = preprocessing.StandardScaler()
standardized_ph = scaler.fit_transform(feature_ph)
print("Standardized pH (Mean and Std):")
print(f"Mean: {round(standardized_ph.mean())}")
print(f"Std: {standardized_ph.std()}")

# Robust Scaler on 'pH'
robust_scaler = preprocessing.RobustScaler()
robust_ph = robust_scaler.fit_transform(feature_ph)
print("\nRobust Scaled pH (First 5):")
```

```
print(robust_ph[:5].flatten())

Standardized pH (Mean and Std):
Mean: 0
Std: 1.0

Robust Scaled pH (First 5):
[ 1.05263158 -0.57894737 -0.26315789 -0.78947368  1.05263158]
```

4. Normalizing Observations

```
# Normalize 'alcohol' and 'quality'
features_norm = df[['alcohol', 'quality']].values
normalizer = Normalizer(norm='l2')
normalized_features = normalizer.transform(features_norm)
print("Normalized Alcohol & Quality (First 5 rows):")
print(normalized_features[:5])

Normalized Alcohol & Quality (First 5 rows):
[[0.88287239 0.46961297]
 [0.89076187 0.45447034]
 [0.89076187 0.45447034]
 [0.8528513 0.52215386]
 [0.88287239 0.46961297]]
```

5. Grouping Observations Using Clustering

```
# Group wines into 3 clusters based on 'alcohol' and 'pH'
features_cluster = df[['alcohol', 'pH']].values
clusterer = KMeans(3, random_state=0)
df['cluster_group'] = clusterer.fit_predict(features_cluster)
print("Clustered Groups (First 5 rows):")
print(df[['alcohol', 'pH', 'cluster_group']].head())

Clustered Groups (First 5 rows):
   alcohol    pH  cluster_group
0      9.4  3.51              1
1      9.8  3.20              1
2      9.8  3.26              1
3      9.8  3.16              1
4      9.4  3.51              1
```

6. Handling Missing Numerical Values

```
# Example: create missing values in 'pH' and impute with mean
df_missing = df.copy()
df_missing.loc[0:10, 'pH'] = np.nan
imputer_mean = SimpleImputer(strategy="mean")
imputed_ph = imputer_mean.fit_transform(df_missing[['pH']])
print("Imputed pH (First 5 values - originally NaNs):")
print(imputed_ph[:5].flatten())

Imputed pH (First 5 values - originally NaNs):
[3.31085642 3.31085642 3.31085642 3.31085642 3.31085642]
```

Part 2: Handling Categorical Data & Imbalanced Classes

7. Imports for Categorical Data

```
from sklearn.preprocessing import LabelBinarizer
from sklearn.feature_extraction import DictVectorizer
from sklearn.neighbors import KNeighborsClassifier
```

8. Encoding Nominal Categorical Features

```
# Example: Assume 'quality' is categorical. Replace with your actual categorical column if different.
feature_quality = df['quality'].values
onehot = LabelBinarizer()
quality_encoded = onehot.fit_transform(feature_quality)
print("One-Hot Encoded (First 5 rows):")
print(quality_encoded[:5])
print("Classes:", onehot.classes_)
```

```
One-Hot Encoded (First 5 rows):
[[0 1 0 0 0]
 [0 0 1 0 0]
 [0 0 1 0 0]
 [0 0 0 1 0]
 [0 0 1 0 0]]
Classes: [3 4 5 6 7 8]
```

9. Encoding Dictionaries of Features

```
# Example: Assume you have categorical columns 'quality' and 'alcohol_type'
data_dict = df[['quality', 'alcohol']].to_dict(orient='records')
dictvectorizer = DictVectorizer(sparse=False)
features_dict = dictvectorizer.fit_transform(data_dict)
print("Dictionary Vectorized Features (First row):")
print(features_dict[0])
print("Feature Names:", dictvectorizer.get_feature_names_out())
```

```
Dictionary Vectorized Features (First row):
[9.4 5. ]
Feature Names: ['alcohol' 'quality']
```

10. Encoding Ordinal Categorical Features & Binning

```
# Example: Bin 'alcohol' into categories
df['alcohol_group'] = pd.cut(df['alcohol'], bins=[0, 8, 12, 16], labels=['Low', 'Medium', 'High'])
scale_mapper = {'Low': 1, 'Medium': 2, 'High': 3}
df['alcohol_group_encoded'] = df['alcohol_group'].map(scale_mapper)
print("Binned and Encoded Alcohol (First 5 rows):")
print(df[['alcohol', 'alcohol_group', 'alcohol_group_encoded']].head())
```

```
Binned and Encoded Alcohol (First 5 rows):
   alcohol alcohol_group alcohol_group_encoded
0      9.4        Medium                  2
1      9.8        Medium                  2
2      9.8        Medium                  2
3      9.8        Medium                  2
4      9.4        Medium                  2
```

11. Imputing Missing Class Values using KNN

```
# Example: Assume missing values in a nominal column 'quality'
from sklearn.neighbors import KNeighborsClassifier

# For demonstration: Mark first 10 rows as missing
y_knn = df['quality'].copy()
y_knn.iloc[:10] = np.nan
X_knn = df[['alcohol', 'pH']].values

# Train using rows that are not missing
mask = y_knn.notnull()
clf_knn = KNeighborsClassifier(n_neighbors=3, weights='distance')
clf_knn.fit(X_knn[mask], y_knn[mask])

# Predict missing
predicted = clf_knn.predict(X_knn[~mask])
print("Predicted missing quality values:", predicted)
```

```
Predicted missing quality values: [5. 5. 6. 6. 5. 5. 5. 6. 5. 5.]
```

12. Handling Imbalanced Classes

```
# Example: Check target balance in 'quality' (or any target variable)
print("Target Distribution:")
print(df['quality'].value_counts())

# Downsample majority classes for balance
```

```
major_class = df['quality'].mode()[0]
minor_class = df['quality'].value_counts().index[-1]
major_indices = np.where(df['quality'] == major_class)[0]
minor_indices = np.where(df['quality'] == minor_class)[0]

desired_size = len(minor_indices)
downsampled_major_indices = np.random.choice(major_indices, size=desired_size, replace=False)
final_indices = np.hstack((minor_indices, downsampled_major_indices))
df_balanced = df.iloc[final_indices]
print("Balanced Target Distribution:")
print(df_balanced['quality'].value_counts())
```

```
Target Distribution:
quality
5    681
6    638
7    199
4     53
8     18
3     10
Name: count, dtype: int64
Balanced Target Distribution:
quality
3    10
5    10
Name: count, dtype: int64
```