```
%matplotlib inline
import warnings
warnings.simplefilter(action='ignore')
```

Reading Data

```
import pandas as pd
import numpy as np

df = pd.read_csv("your_file.txt",
                 delim_whitespace=True,
                 na_values="?")
df.head()

df = pd.read_csv("your_file.csv",sep=";")
df_xl = pd.read_xls("your_file.xls")

df = pd.read_excel("your_file.xlsx",
sheet_name="Sheet1")
```

Preprocess

```
df.shape / df.columns  / df.info()
df.drop('variable_name',axis=1,inplace= True)
```

Change Variable Name :

```
df.rename(columns={'old_column_name':
'new_column_name'}, inplace=True)

#Convert variable into binary using median
df['mpg']=df['mpg'].apply(lambda x: 'high MPG' if x >
median_mpg else 'low MPG' )
```

Summary stats

```
df.describe()   / df.dtypes
df['column'].value_counts()  / df['column'].nunique()
```

Distribution of variables

```
sns.pairplot(auto_mpg_data,diag_kind='kde',kind='reg',
plot_kws={'line_kws':{'color':'red'}})
```

Duplicates

```
duplicate = df[df.duplicated()]
df = df.drop_duplicates()
```

Merging

```
df_combined = pd.concat([df, df2], axis=0) or 1 for
columns
combined_df = pd.merge(df1, df2, on="id")
```

Missing

```
df.isna().sum()
missing = ['na','.']
na_percentage=df.isna().mean()*100
remain_column=na_percentage[na_percentage <
50].index
df=df[remain_column]
df.dropna(axis=0,inplace=True)# remove all missing
```

Imputation

```
df['Gender']=df['Gender'].fillna(df['Gender'].mode().ilo
c[0])
df['Age'].fillna((round(df['Age'].mean(),0)),inplace=Tr
ue)
df['Cabin'].fillna(df['Cabin'].mode()[0],inplace=True)

df1 = df.fillna(method="ffill")
df2 = df.fillna(method="bfill")
zero_imputed_data = data.fillna(0)

df_copy['smokingStatus'] =
df_copy['smokingStatus'].map({'YES': 1, 'NO': 0})
```

Outliers

```
def handle_outliers(column, threshold=1.5):
   Q1 = column.quantile(0.25)
   Q3 = column.quantile(0.75)
   IQR = Q3 - Q1

   lower_bound = Q1 - threshold * IQR
   upper_bound = Q3 + threshold * IQR

   # Replace outliers with the median value
   median = column.median()
   column = np.where((column < lower_bound) |
(column > upper_bound), median, column)

   return column

numerical_column = df["impact.significance"]

df["impact.significance"] =
handle_outliers(numerical_column)
df.head()

or
numerical_columns =
data.select_dtypes(include=[np.number])

outlier_handled_data =
numerical_columns.apply(handle_outliers)

or
Q1 = data.quantile(0.25)
Q3 = data.quantile(0.75)
IQR = Q3 - Q1
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR
outliers = ((data < lower_bound) | (data >
upper_bound))
outliers.sum()
```

Scaling

```
scaler = MinMaxScaler/StandardScaler()
scaled data = scaler.fit_transform(data)
```

scaled_data = pd.DataFrame(scaled data, columns=data.columns)
df[["x1", "x2"]] = scaler.fit_transform(df[["x1", "x2"]])

Data Extraction
selected_columns = data[['Name','Salary']]
rec = df.iloc[3]

Re-shaping using melt()
melted_data=data.melt(id_vars="Country",var_name="Quarter",value_name="Value")

Dummy Coding
dummy_coded_data=pd.get_dummies(data['Color'],dtype = int)
or
df = pd.get_dummies(df, columns=['dummy_cols'], drop_first=True)

One Hot Encoder
from sklearn.preprocessing import OneHotEncoder
encoder =OneHotEncoder()
dummy_coded_data=encoder.fit_transform(data[['Color','Size']])
# Convert the result to a DataFrame
dummy_coded_df=pd.DataFrame(dummy_coded_data.toarray(),columns=encoder.get_feature_names_out(['Color','Size']))

Label Encoder
from sklearn.preprocessing import LabelEncoder
encoder = LabelEncoder()
df['Color'] = encoder.fit_transform(df['Color'])

Change data type
df['Gender'] = df['Gender'].astype('category')

<u>EDA</u>

import matplotlib.pyplot as plt
import seaborn as sns

#Getting unique values and their counts in each column
unique_values_counts = {col: df[col].value_counts() for col in df.columns}
for col, value_counts in unique_values_counts.items():
    print(f"\nColumn '{col}':")
    print(f"Unique values and their counts in column '{col}':")
    print("\n",value_counts)
    print()

#Summary statistics
print(df.describe())
print(df['column'].value_counts())
print(df['column'].unique())

#Visualizations
plt.figure(figsize=(10,6))
df.groupby(['category']).size().plot(kind='pie',labels = ['x', 'y'])

categorical_columns = df.select_dtypes(include=['object', 'category'])
numerical_columns = df.select_dtypes(include=['number']).columns.tolist()

for column in numeric_col:
    sns.histplot(data=df, x=column, bins=10, kde=True)
    plt.show()

sns.pairplot(df, diag_kind='kde', kind='reg', plot_kws={'line_kws':{'color':'red'}})
sns.histplot(df['column'], bins=10, kde=True)
sns.boxplot(x='column', y='target', data=df)
sns.scatterplot(x='x_col', y='y_col', data=df)
sns.heatmap(df.corr(), annot=True, cmap='coolwarm')
sns.countplot(x='category', data=df)

Pie Chart
for column in categorical_columns:
    plt.figure(figsize=(8, 8))
    df[column].value_counts().plot(kind='pie', autopct='%1.1f%%', startangle=90)
    plt.title(f'Pie Chart of {column}')
    plt.ylabel('')
    plt.show()

Save the edited dataset as a CSV
df.to_csv('df.csv', index = False)

Create Random Numbers
np.random.seed(42)
x = np.random.rand(100)
# dir(np.random) # what are the function available in random
epsilon = np.random.normal(0,0.25,100)
y = 2+3*x + epsilon

Split into train test sets
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state=1)

Simple Linear Regression
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.impute import SimpleImputer

# Impute missing values with the mean
imputer = SimpleImputer(strategy='mean')
X_imputed = imputer.fit_transform(X_encoded)

```
X_imputed = pd.DataFrame(X_imputed,
columns=X_encoded.columns)

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

y_pred = model.predict(X_test)

rmse = np.sqrt(mean_squared_error(y_test, y_pred))
print(f'RMSE: {rmse}')

r_squared = r2_score(y_test, y_pred)

# Get the intercept
intercept = model.intercept_

# Get the coefficients
coefficients = model.coef_
```

Ridge /Lasso Regression

```
from sklearn.linear_model import Lasso, LassoCV
from sklearn.linear_model import Ridge, RidgeCV
from sklearn.metrics import mean_squared_error

alphas = 10**np.linspace(10, -2, 100)
ridge_cv = RidgeCV(alphas=alphas,
store_cv_values=True)
#lasso = Lasso(max_iter=10000)
ridge_cv.fit(X_train, y_train)

best_alpha = ridge_cv.alpha_

ridge_best = Ridge(alpha=best_alpha)
ridge_best.fit(X_train, y_train)

ridge_pred = ridge_best.predict(X_test)
mse = mean_squared_error(y_test, ridge_pred)

ridge_coef = ridge_best.coef_

num_features = len(ridge_coef)
num_nonzero_features = np.sum(ridge_coef != 0)
```

Class Imbalanced

```
from collections import Counter
from imblearn.under_sampling import
RandomUnderSampler
from imblearn.over_sampling import
RandomOverSampler, SMOTE
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report,
confusion_matrix

print("Original class distribution:", Counter(y))
```

<u>Under-Sampling / Over sampling</u>

```
under_sampler =
RandomUnderSampler(random_state=42)
#RandomOverSampler
X_resampled_under, y_resampled_under =
under_sampler.fit_resample(X_train, y_train)  ##Over
print(Counter(y_resampled_under))
```

<u>SMOTE</u>

```
smote = SMOTE(random_state=42)
X_resampled_smote, y_resampled_smote =
smote.fit_resample(X_train, y_train)
print("SMOTE class distribution:",
Counter(y_resampled_smote))

def plot_class_distribution(y, title):
   plt.figure(figsize=(6,4))
   plt.bar(Counter(y).keys(), Counter(y).values(),
color=['blue', 'orange'])
   plt.title(title)
   plt.xlabel('Class')
   plt.ylabel('Number of instances')
   plt.show()
```

<u>Building and Evaluating Models</u>

```
def evaluate_model(X_train, y_train, X_test, y_test):
   model = LogisticRegression(random_state=42)
   model.fit(X_train, y_train)
   y_pred = model.predict(X_test)
   print(confusion_matrix(y_test, y_pred))
   print(classification_report(y_test, y_pred))
```

PCA

```
from sklearn.decomposition import PCA
from sklearn.impute import SimpleImputer

# Perform PCA, keeping the first 10 components
pca = PCA(n_components=5)
pca_result = pca.fit_transform(x_scaled)

# Proportion of variance explained by each component
prop_var = pca.explained_variance_ratio_
PC_numbers = np.arange(pca.n_components_) + 1

# Scree Plot
plt.plot(PC_numbers, prop_var, 'mo-')
plt.title('Scree Plot', fontsize=12)
plt.ylabel('Proportion of Variance', fontsize=8)
plt.xlabel('No of Components')
plt.show()

# Cumulative variance explained
cum_var = pca.explained_variance_ratio_.cumsum()

# Loadings (PCA components)
loadings = pca.components_
```

```
feature_names = x_cols

# Get the loadings for the first two principal
components (PC1 and PC2)
xs = loadings[0]
ys = loadings[1]

# Plot the loadings on a scatterplot
plt.figure(figsize=(8, 6))
for i, varnames in enumerate(feature_names):
   plt.scatter(xs[i], ys[i], s=100)
   plt.arrow(
      0, 0,  # coordinates of arrow base
      xs[i],  # length of the arrow along x
      ys[i],  # length of the arrow along y
      color='pink',
      head_width=0.005
   )
   plt.text(xs[i], ys[i], varnames, fontsize=11)
# Define the axes
xticks = np.linspace(-0.3, 0.4, num=5)
yticks = np.linspace(-0.4, 0.6, num=5)
plt.xticks(xticks)
plt.yticks(yticks)
# Label the axes with the proportion of variance
explained
plt.xlabel(f'PC1
({pca.explained_variance_ratio_[0]*100:.1f}%)')
plt.ylabel(f'PC2
({pca.explained_variance_ratio_[1]*100:.1f}%)')
plt.title('Loadings Plot', fontsize=14)
plt.show()
```

Clustering

```
from sklearn.cluster import KMeans, DBSCAN
kmeans = KMeans(n_clusters=3, random_state=42)
kmeans.fit(X)
clusters = kmeans.predict(X)
dbscan = DBSCAN(eps=0.5, min_samples=5)
dbscan.fit(X)
labels = dbscan.labels_
```

Classification

```
from sklearn.linear_model import LogisticRegression,
Ridge, Lasso, ElasticNet
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier,
AdaBoostClassifier
from sklearn.svm import SVC
from xgboost import XGBClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, f1_score,
precision_score, recall_score, confusion_matrix,
roc_auc_score, roc_curve, auc, classification_report,
precision_recall_curve, ConfusionMatrixDisplay
from sklearn.model_selection import train_test_split,
cross_val_score, StratifiedKFold, GridSearchCV,
RandomizedSearchCV
from sklearn.preprocessing import MinMaxScaler,
LabelEncoder, StandardScaler, RobustScaler
import xgboost as xgb
```

 # KNN Classifier

```
knn_model =
KNeighborsClassifier().fit(X_train,y_train)
y_pred = knn_model.predict(X_test)

print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test,y_pred))
```

#RF Classifier

```
rf = RandomForestClassifier(random_state = 42,
max_depth = 5)
rf.fit(X_train, y_train)
y_pred = rf.predict(X_test)
```

 # RandomForestRandom Grid Search CV

```
param_distributions = {
    'n_estimators': np.arange(10, 200, 10),
'max_depth': [None] + list(np.arange(5, 30, 5)),
 }

# Create the RandomizedSearchCV object
random_search = RandomizedSearchCV(rf,
param_distributions, n_iter=100, cv=5,
random_state=42)
random_search.fit(X_train, y_train)

# Get the best hyperparameter values
best_params = random_search.best_params_
print("Best Hyperparameters: ", best_params)
```

 # Hyper parameter-tuned RF

```
best_rf = RandomForestClassifier(random_state=42,
        n_estimators=best_params['n_estimators'],
        max_depth=best_params['max_depth'],
        min_samples_split=best_params['min_samples
        _split'],
        min_samples_leaf=best_params['min_samples_
        leaf'],
        max_features=best_params['max_features'])

# Confusion matrix display
cm = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix =
cm)
plt.figure(figsize=(5, 5))
disp.plot(values_format='.0f')
plt.grid(False)
plt.show()
```

Decision Tree

```
model = DecisionTreeClassifier(random_state=42)
# Fit the model to the training data
model.fit(X_train, y_train)
```

Naive Bayes

```
model1 = GaussianNB()
model2 = MultinomialNB() # Create a Multinomial Naive Bayes Classifier (for discrete data)
model1.fit(trainx, trainy)
model2.fit(trainx , trainy)
```

SVM

```
model = svm.SVC(kernel='linear') #linearly seperable
# Fit the classifier to the training data
model.fit(X_train, y_train)
```

XG Boost

```
# pip install xgboost

import xgboost as xgb
from sklearn.model_selection import train_test_split
from sklearn.metrics import log_loss

# Create DMatrix for XGBoost
dtrain = xgb.DMatrix(X_train, label=y_train)
dtest = xgb.DMatrix(X_test, label=y_test)

# Set parameters for XGBoost
params = {
    'max_depth': 3,
    'eta': 0.1,
    'objective': 'binary:logistic',  # For binary classification
    'eval_metric': 'logloss'  # Use logloss as the evaluation metric
}

# Train the model
num_rounds = 100
bst = xgb.train(params, dtrain, num_rounds)

# Predict the probabilities for the test set
y_pred_prob = bst.predict(dtest)

# If you want to evaluate the model, you can calculate log loss or other metrics
log_loss_value = log_loss(y_test, y_pred_prob)
print(f"Log Loss: {log_loss_value}")
```

Regression problem models and performance metrics

```
from sklearn.linear_model import LinearRegression, Ridge, Lasso, ElasticNet
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
import xgboost as xgb

models = {
'Linear Regression': LinearRegression(),
'Ridge': Ridge(alpha=1.0),
'Lasso': Lasso(alpha=1.0),
'Elastic-net':ElasticNet(alpha=1.0, l1_ratio=0.5),
'GausianNB':GaussianNB(),
'SVC': SVC(probability=True),
'XG Boosting': xgb.XGBRegressor(n_estimators=100, random_state=42),
'rf_model': RandomForestRegressor(n_estimators=100, random_state=42)

}

def calculate_mape(y_true, y_pred):
    return np.mean(np.abs((y_true - y_pred) / y_true)) * 100

for name, model in models.items():
    model.fit(X_train, y_train)

    # Training set predictions
    y_train_pred = model.predict(X_train)
    train_mse = mean_squared_error(y_train, y_train_pred)
    train_r2 = r2_score(y_train, y_train_pred)
    rmse_train = np.sqrt(train_mse)
    mape_train = calculate_mape(y_train, y_train_pred)

    # Test set predictions
    y_test_pred = model.predict(X_test)
    test_mse = mean_squared_error(y_test, y_test_pred)
    test_r2 = r2_score(y_test, y_test_pred)
    rmse_test = np.sqrt(test_mse)
    mape_test = calculate_mape(y_test, y_test_pred)

    # Print results
    print(name)
    print('='*len(name))
    print(f"Train Mean Squared Error: {train_mse}")
    print(f"Train R-squared: {train_r2}")
    print(f"Train RMSE: {rmse_train:.2f}")
    print(f"Train MAPE: {mape_train:.2f}%")

    print('\n')
    print(f"Test Mean Squared Error: {test_mse}")
    print(f"Test R-squared: {test_r2}")
    print(f"Test RMSE: {rmse_test:.2f}")
    print(f"Test MAPE: {mape_test:.2f}%")
    print('\n\n')
```

Binary Classification problem models and performance metrics

```
models = {
'Multiple Logistic': LogisticRegression(),
```

```
'Ridge': LogisticRegression(penalty='l2', C=1.0),
'Lasso': LogisticRegression(penalty='l1', C=1'),
'Elastic-net':LogisticRegression(penalty='elasticnet',
l1_ratio=0.5, C=1),
'Random Forest': RandomForestClassifier(),
'GausianNB':GaussianNB(),
'SVC': SVC(probability=True),
}

for name, model in models.items():
model.fit(train_feature, train_label)
 # Training set
train_pred = model.predict(train_feature)
train_acc = accuracy_score(train_label, train_pred)
train_err = 1 - train_acc
train_f1 = f1_score(train_label, train_pred,
average='weighted')
train_cm = confusion_matrix(train_label, train_pred)
train_sensitivity = train_cm[1, 1] / (train_cm[1, 1] +
train_cm[1, 0])
train_specificity = train_cm[0, 0] / (train_cm[0, 0] +
train_cm[0, 1])

# Test set
testpred = model.predict(test_feature)
test_acc = accuracy_score(test_label, test_pred)
test_err = 1 - test_acc
test_f1 = f1_score(test_label, test_pred,
average='weighted') test_cm =
confusion_matrix(test_label, test_pred) test_sensitivity
= test_cm[1, 1] / (test_cm[1, 1] + test_cm[1, 0])
test_specificity = test_cm[0, 0] / (test_cm[0, 0] +
test_cm[0, 1])

print(name) print('='*len(name))
print(f'Training Accuracy: {train_acc:.4f}')
print(f'Training Error Rate: {train_err:.4f}')
print(f'Training F1 Score: {train_f1:.4f}')
print(f'Training Confusion Matrix:\n{train_cm}')
print("Training sensitivity: ", train_sensitivity)
print("Training specificity: ", train_specificity)
print('\n')
print(f'Test Accuracy: {test_acc:.4f}')
print(f'Test Error Rate: {test_err:.4f}')
print(f'Test F1 Score: {test_f1:.4f}')
print(f'Test Confusion Matrix:\n{test_cm}')
print("Test sensitivity: ", test_sensitivity)
print("Test specificity: ", test_specificity)

print('\n\n')
```

multinomial Classification problem models

```
models = {
'logistic':LogisticRegression(solver='lbfgs',
multi_class='multinomial'),
'Ridge': LogisticRegression(solver='saga',
multi_class='multinomial',penalty='l2', C=1.0),
'Lasso': LogisticRegression(solver='saga',
multi_class='multinomial',penalty='l1', C=1),
'Elastic-net':LogisticRegression(penalty='elasticnet',
l1_ratio=0.5, C=1, solver='saga',
multi_class='multinomial')

## solver='saga' for large datasets, 'liblinear' for small
datasets

}
```

Roc curve

```
for name, model in models.items():
    model.fit(train_feature, train_label)


    # ROC curve and AUC
    if hasattr(model, 'predict_proba'): # Check if
    the model supports predict_proba
    test_pred_proba=model.predict_proba(test_feat
    ure)[:, 1]
    else:
    test_decision =
    model.decision_function(test_feature)
    test_pred_proba = 1 / (1 +
    np.exp(-test_decision)) # Convert decision
    values to probabilities

    fpr, tpr, _ = roc_curve(test_label,
    test_pred_proba) roc_auc = auc(fpr, tpr)

    # Plot ROC curve
    plt.figure()
    plt.plot(fpr, tpr, color='darkorange', lw=2,
    label=f'ROC curve (AUC = {roc_auc:.2f})')
    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(f'ROC Curve - {name}')
    plt.legend(loc='lower right')
    plt.show()
```

Assumption Checking

```
y_train_pred = model.predict(X_train)

## Linearity

# Residuals
residuals = y_train - y_train_pred

# Scatter plot of residuals vs. predicted values
plt.figure(figsize=(10, 6))
sns.scatterplot(x=y_train_pred, y=residuals)
```

```
plt.axhline(0, color='red', linestyle='--')
plt.xlabel('Predicted Values')
plt.ylabel('Residuals')
plt.title('Residuals vs. Predicted Values')
plt.show()

## Independence
from statsmodels.stats.stattools import durbin_watson

# Calculate Durbin-Watson statistic
dw_stat = durbin_watson(residuals)

## Normality
import scipy.stats as stats

# Q-Q plot
plt.figure(figsize=(10, 6))
stats.probplot(residuals, dist="norm", plot=plt)
plt.title('Q-Q Plot')
plt.show()

# Histogram of residuals
plt.figure(figsize=(10, 6))
sns.histplot(residuals, kde=True)
plt.xlabel('Residuals')
plt.title('Distribution of Residuals')
plt.show()
```

Remove Multicollinearity

```
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import statsmodels.api as sm
from statsmodels.stats.outliers_influence import
variance_inflation_factor

# Load and prepare the data
# Assuming 'data' is a pandas DataFrame with the
necessary columns
data_standardized =
StandardScaler().fit_transform(data[['FatPercentage',
'Weight', 'PhysicalActivity', 'Thickness']])

# Perform PCA
pca = PCA()
pca_result = pca.fit_transform(data_standardized)

# Scree plot (explained variance)
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 6))
plt.plot(range(1, len(pca.explained_variance_ratio_) +
1), pca.explained_variance_ratio_, marker='o')
plt.title('Scree Plot')
plt.xlabel('Principal Component')
plt.ylabel('Variance Explained')
plt.show()

# Number of components to retain (90% of variance)
cumulative_variance =
pca.explained_variance_ratio_.cumsum()
num_components = next(i for i, total_variance in
enumerate(cumulative_variance) if total_variance >=
0.90) + 1

print(f"Number of principal components to retain:
{num_components}")

# Extract the selected principal components
pc_data = pd.DataFrame(pca_result[:,
:num_components])

# Fit the linear regression model using principal
components
X = sm.add_constant(pc_data)
y = data['Density']

pca_model = sm.OLS(y, X).fit()
print(pca_model.summary())

# Calculate VIF for the PCA model
vif_data = pd.DataFrame()
vif_data['feature'] = X.columns

# For each feature, calculate VIF
vif_data['VIF'] = [variance_inflation_factor(X.values,
i) for i in range(X.shape[1])]
print(vif_data)
```

GVIF

```
from statsmodels.stats.outliers_influence import
variance_inflation_factor
import statsmodels.api as sm
#X = df[['variable1', 'variable2', 'variable3']]
X = sm.add_constant(df2)
vif = pd.DataFrame()
vif['Variable'] = X.columns
vif['VIF'] = [variance_inflation_factor(X.values, i) for i
in range(X.shape[1])]
# Calculate GVIF vif['GVIF'] = np.sqrt(vif['VIF'])
print(vif)

vif['GVIF'] = np.sqrt(vif['VIF'])
print(vif)
```

VIF

```
import pandas as pd
from statsmodels.stats.outliers_influence import
variance_inflation_factor
import statsmodels.api as sm

X = df.drop(columns=['DependentVariable'])
```

```
# Add a constant (intercept) to the model if not included already
X = sm.add_constant(X)

# Calculate VIF for each feature
vif_data = pd.DataFrame()
vif_data['feature'] = X.columns
vif_data['VIF'] = [variance_inflation_factor(X.values, i) for i in range(X.shape[1])]

print(vif_data)
```

<u>Neural Networks</u>

<u>Feed forward Neural Networks</u>

Binary Class

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
import keras

X=df.drop(columns=["Target"])
y=df["Target"]

from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test =
train_test_split(X_scaled, y, test_size=0.2,
random_state=123)

#changing response varible y to one hot vectors
y_train=keras.utils.to_categorical(y_train)
y_test=keras.utils.to_categorical(y_test)

# Build the model
model = Sequential([ Dense(64, activation='relu',
input_shape=(X_train.shape[1],)), Dense(64,
activation='relu'),Dense(64, activation='relu'),
Dense(2, activation='sigmoid')
])
model.summary()

model.compile(optimizer=Adam(),
loss='binary_crossentropy', metrics=['accuracy'])

model.fit(X_train, y_train, epochs=20, batch_size=8,
validation_split=0.2)

loss, accuracy = model.evaluate(X_test, y_test)
print(f'Accuracy: {accuracy:.2f}')

pred_array=model.predict(X_test)
predicted_classes = np.argmax(pred_array, axis=1)
```

Multi Class

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
import keras

X=df.drop(columns=["Target"])
y=df["Target"]

y=y.astype('object')


from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test =
train_test_split(X_scaled, y, test_size=0.2,
random_state=123)

#changing response varible y to one hot vectors
y_train=keras.utils.to_categorical(y_train)
y_test=keras.utils.to_categorical(y_test)

model = Sequential([ Dense(64, activation='relu',
input_shape=(X_train.shape[1],)), Dense(64,
activation='relu'),Dense(64, activation='relu'),
Dense(3, activation='softmax')
])

model.summary()

model.compile(optimizer=Adam(),
loss='categorical_crossentropy', metrics=['accuracy'])

model.fit(X_train, y_train, epochs=20, batch_size=8,
validation_split=0.2)

loss, accuracy = model.evaluate(X_test, y_test)
print(f'Accuracy: {accuracy:.2f}')

pred_array=model.predict(X_test)
predicted_classes = np.argmax(pred_array, axis=1)

predicted_classes
```

Categorical datasets, Onehot-encoding and scaling

```
cats=['Category','Category1']
quants=['Feature1','Feature2','Feature3','Feature4']

from sklearn.preprocessing import OneHotEncoder
encoder =OneHotEncoder()
```

```
dummy_coded_data = encoder.fit_transform(df[cats])

dummy_coded_df=pd.DataFrame(dummy_coded_data.toarray(),columns=encoder.get_feature_names_out(cats))

X=df.drop(columns=["Target"])
y=df["Target"]

y=y.astype('object')

from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_numerical=df[quants]
X_scaled_numeric = scaler.fit_transform(X_numerical)

cat_array=np.array(dummy_coded_df)

X=np.concatenate((X_scaled_numeric, cat_array), axis=1)

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=123)
```

FNN (Regression case)

```
X=df.drop(columns=["Response"])
y=df["Response"]

from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X_scaled , y, test_size=0.2, random_state=123)

model = Sequential([
   Dense(64, activation='relu', input_shape=(X_train.shape[1],)),  # Input layer
   Dense(32, activation='relu'),  # Hidden layer
   Dense(1)  # Output layer for regression
])

model.summary()

model.compile(optimizer=Adam(), loss='mean_squared_error')

model.fit(X_train, y_train, epochs=20, batch_size=8, validation_split=0.2)

loss = model.evaluate(X_test, y_test)
print(f'Test loss: {loss}')

y_pred = model.predict(X_test)
```

activation function

```
import numpy as np

def sigmoid(x):
   return 1 / (1 + np.exp(-x))

def relu(x):
   return np.maximum(0, x)

# Example usage
x = np.array([-1, 0, 1])
print("Sigmoid:", sigmoid(x))
print("ReLU:", relu(x))
```