Regression

```
import pandas as pd
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor,
GradientBoostingRegressor
from sklearn.linear_model import LinearRegression, Ridge
from sklearn.metrics import mean_squared_error, r2_score,
mean_absolute_error, explained_variance_score
from sklearn.feature_selection import SelectKBest, f_regression

# Load data
data = pd.read_csv('/content/Ames_Housing_Data.csv')

# Splitting data into features and target
X = data.drop('SalePrice', axis=1)
y = data['SalePrice']

# Identifying numeric and categorical columns
numeric_features = X.select_dtypes(include=['int64', 'float64']).columns
categorical_features = X.select_dtypes(include=['object']).columns

# Splitting the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Creating a Column Transformer for scaling and encoding
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='mean',fill_value='missing')),
    ('scaler', StandardScaler())])

categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='constant', fill_value='missing')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))])

preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)])

# Feature Selection
feature_selection = SelectKBest(score_func=f_regression, k='all')

# Model: RandomForestRegressor
rf_model = Pipeline(steps=[
```

```python
    ('preprocessor', preprocessor),
    ('feature_selection', feature_selection),
    ('model', RandomForestRegressor(n_estimators=100, random_state=42))])

# Training the RandomForestRegressor
rf_model.fit(X_train, y_train)

# Making predictions with RandomForestRegressor
y_pred_rf = rf_model.predict(X_test)

# Evaluation of RandomForestRegressor
print("RandomForestRegressor Evaluation:")
print("Mean Squared Error:", mean_squared_error(y_test, y_pred_rf))
print("R^2 Score:", r2_score(y_test, y_pred_rf))
print("Mean Absolute Error:", mean_absolute_error(y_test, y_pred_rf))
print("Explained-Variance-Score:",explained_variance_score(y_test, 
y_pred_rf))
print("\n")
```

Classification

```
import pandas as pd
! pip install -q shap
import shap
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report,
roc_auc_score
from imblearn.over_sampling import SMOTE
from imblearn.pipeline import Pipeline as ImbPipeline
from sklearn.feature_selection import SelectKBest, f_classif




# Load data
data = pd.read_csv('/content/Telco-Customer-Churn.csv')

# Splitting data into features and target
X = data.drop('Churn', axis=1)
y = data['Churn']

# Splitting the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Identifying numeric and categorical columns
numeric_features = X.select_dtypes(include=['int64',
'float64']).columns
categorical_features = X.select_dtypes(include=['object',
'category']).columns

# Numeric Transformer
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='mean')),
    ('scaler', StandardScaler())])

# Categorical Transformer
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent',
fill_value='missing')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))])

# Column Transformer
```

```
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)])

# Feature Selection
feature_selection = SelectKBest(score_func=f_classif, k='all')  # Use a
suitable value for k

# SMOTE for handling class imbalance
smote = SMOTE(random_state=42)

# RandomForestClassifier model
rf_classifier = RandomForestClassifier(random_state=42)

# Pipeline with preprocessing, feature selection, SMOTE, and model
model = ImbPipeline(steps=[('preprocessor', preprocessor),
                           ('feature_selection', feature_selection),
                           ('smote', smote),
                           ('classifier', rf_classifier)])

# Training the model
model.fit(X_train, y_train)

# Predicting the test set results
y_pred = model.predict(X_test)
y_proba = model.predict_proba(X_test)[:, 1]  # Probabilities for ROC-
AUC

# Evaluating the model
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred)
roc_auc = roc_auc_score(y_test, y_proba)

print(f'Accuracy: {accuracy}')
print(f'Classification Report: \n{report}')
print(f'ROC-AUC Score: {roc_auc}')

# Feature Importance
feature_importances = rf_classifier.feature_importances_


# Plotting Feature Importances
feature_importances = rf_classifier.feature_importances_
sorted_idx = feature_importances.argsort()

plt.figure(figsize=(10, 10))
plt.title("Feature Importances")
```

Recommendation System- Collaborative Filtering (SVD, NCF)

SVD explanation-

1. **SVD Decomposition:**

   pythonCopy code

   - The function uses the `svds` method from `scipy.sparse.linalg` to perform the singular value decomposition on the `ratings_matrix`.
   - It decomposes `ratings_matrix` into three matrices: `u`, `sigma`, and `vt`.
   - `u` is a matrix with users on rows and latent factors on columns.
   - `sigma` is a vector of singular values (not a matrix).
   - `vt` (V transpose) is a matrix with items on rows and latent factors on columns.
   - The number of latent factors retained is specified by `k`.

2. **Converting Sigma to a Diagonal Matrix:**

   pythonCopy code

   - This line converts the vector of singular values `sigma` into a diagonal matrix.
   - In the SVD, the sigma vector represents the strength of each latent factor. For collaborative filtering, we convert this vector into a diagonal matrix to be used in the reconstruction of the approximated ratings matrix.

3. **Reconstructing the Approximated Ratings Matrix:**

   pythonCopy code

   - This line reconstructs the approximated ratings matrix from the decomposed matrices `u`, `sigma`, and `vt`.
   - The `np.dot` function is used for matrix multiplication.
   - The reconstructed matrix `predicted_ratings` is an approximation of the original `ratings_matrix`, but now it also contains predictions (estimated ratings) for previously missing values.

Collaborative Rec System Code

```
# Import necessary libraries
import numpy as np
import pandas as pd
import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, mean_absolute_error
from scipy.sparse.linalg import svds
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Embedding, Flatten, Dense,
Concatenate, Dropout
from tensorflow.keras.regularizers import l2
from tensorflow.keras.callbacks import EarlyStopping,
LearningRateScheduler

# Load the dataset with specified data type for the column
df = pd.read_csv('/content/rating.csv', dtype={3: str})

#df = df[['userId','movieId', 'rating']]

# Data Preprocessing
df = df.dropna()  # Drop missing values
num_users = df['userId'].nunique()
num_items = df['movieId'].nunique()

# User-item matrix for SVD
user_item_matrix = df.pivot(index='userId', columns='movieId',
values='rating').fillna(0)

# Convert user and item IDs to categorical codes for NCF
from sklearn.preprocessing import LabelEncoder

# Create label encoders for user and item IDs
user_encoder = LabelEncoder()
item_encoder = LabelEncoder()

# Fit and transform the user and item IDs to integer labels
user_ids = user_encoder.fit_transform(df['userId'])
item_ids = item_encoder.fit_transform(df['movieId'])
ratings = df['rating'].values

X = np.column_stack((user_ids, item_ids))
y = ratings
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# SVD Model
```

```
def svd_collaborative_filtering(ratings_matrix, k=50):
    u, sigma, vt = svds(ratings_matrix, k=k)
    sigma = np.diag(sigma)
    predicted_ratings = np.dot(np.dot(u, sigma), vt)
    return predicted_ratings

# Neural Collaborative Filtering (NCF) Model with Enhanced Layers
def create_ncf_model(num_users, num_items, embedding_size=20):
    user_input = Input(shape=(1,))
    item_input = Input(shape=(1,))
    user_embedding = Embedding(num_users, embedding_size,
input_length=1)(user_input)
    item_embedding = Embedding(num_items, embedding_size,
input_length=1)(item_input)
    user_flatten = Flatten()(user_embedding)
    item_flatten = Flatten()(item_embedding)
    concat = Concatenate()([user_flatten, item_flatten])
    dense1 = Dense(128, activation='relu',
kernel_regularizer=l2(0.001))(concat)
    dropout1 = Dropout(0.05)(dense1)
    dense2 = Dense(64, activation='relu')(dropout1)
    dropout2 = Dropout(0.03)(dense2)
    output = Dense(1, activation='linear')(dropout2)
    model = Model(inputs=[user_input, item_input], outputs=output)
    model.compile(optimizer='adam', loss='mean_squared_error')
    return model

# Early Stopping and Learning Rate Scheduler for NCF
early_stopping = EarlyStopping(monitor='val_loss', patience=3)
lr_scheduler = LearningRateScheduler(lambda epoch, lr: lr * 0.9 if
epoch > 2 else lr)

# Training the NCF Model
# Split user IDs and item IDs in training and testing sets
X_train_users, X_train_items = X_train[:, 0], X_train[:, 1]
X_test_users, X_test_items = X_test[:, 0], X_test[:, 1]

# Convert to tf.data.Dataset and structure for two inputs
train_dataset = tf.data.Dataset.from_tensor_slices(((X_train_users,
X_train_items), y_train))
test_dataset = tf.data.Dataset.from_tensor_slices(((X_test_users,
X_test_items), y_test))

# Apply shuffle and batch
train_dataset = train_dataset.shuffle(buffer_size=1000).batch(128)
test_dataset = test_dataset.batch(128)

# Training the NCF Model
```

```python
ncf_model = create_ncf_model(num_users, num_items)
ncf_model.fit(train_dataset, epochs=5, callbacks=[early_stopping,
lr_scheduler])

# Predictions and Evaluation for NCF Model
y_pred_ncf = ncf_model.predict(test_dataset).flatten()
ncf_rmse = np.sqrt(mean_squared_error(y_test, y_pred_ncf))
ncf_mae = mean_absolute_error(y_test, y_pred_ncf)


# Using SVD Model
predicted_ratings_svd =
svd_collaborative_filtering(user_item_matrix.values)
actual_ratings = user_item_matrix.values[user_item_matrix.notna()]

# Flatten the matrices for comparison
predicted_ratings_svd_flat =
predicted_ratings_svd.flatten()[user_item_matrix.notna().values.flatten
()]
actual_ratings_flat = actual_ratings.flatten()

# Evaluate SVD Model
def rmse(y_true, y_pred):
    return np.sqrt(mean_squared_error(y_true, y_pred))

def mae(y_true, y_pred):
    return mean_absolute_error(y_true, y_pred)

svd_rmse = rmse(actual_ratings_flat, predicted_ratings_svd_flat)
svd_mae = mae(actual_ratings_flat, predicted_ratings_svd_flat)

# Print Evaluation Results
print("SVD RMSE:", svd_rmse)
print("SVD MAE:", svd_mae)
print("\n###########################################\n")
print("NCF RMSE:", ncf_rmse)
print("NCF MAE:", ncf_mae)
```

Recommendation System- Collaborative Filtering (NCF with textual data)

```
import numpy as np
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Embedding, Flatten, Dense,
Concatenate, Dropout
from tensorflow.keras.regularizers import l2

# Example DataFrame
df = pd.DataFrame({
    'userId': np.random.randint(1, 100, 1000),
    'movieId': np.random.randint(1, 500, 1000),
    'rating': np.random.randint(1, 6, 1000),
    'product_description': np.random.choice(['Action', 'Drama',
'Comedy', 'Thriller'], 1000)
})

# Preprocessing Text Data
tfidf = TfidfVectorizer(max_features=100)
tfidf_matrix = tfidf.fit_transform(df['product_description']).toarray()

# Data Preparation for NCF
user_ids = df['userId'].astype("category").cat.codes.values
item_ids = df['movieId'].astype("category").cat.codes.values
ratings = df['rating'].values

X = np.column_stack((user_ids, item_ids, tfidf_matrix))
y = ratings
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

num_users, num_items = len(df['userId'].unique()),
len(df['movieId'].unique())

# NCF Model with Text Feature
def create_ncf_model_with_text(num_users, num_items, text_features_dim,
embedding_size=20):
    # User, Item, and Text Input Layers
    user_input = Input(shape=(1,))
    item_input = Input(shape=(1,))
    text_input = Input(shape=(text_features_dim,))

    # Embeddings
```

```
    user_embedding = Embedding(num_users, embedding_size,
input_length=1)(user_input)
    item_embedding = Embedding(num_items, embedding_size,
input_length=1)(item_input)

    # Flatten the embeddings
    user_flatten = Flatten()(user_embedding)
    item_flatten = Flatten()(item_embedding)

    # Concatenate all inputs
    concat = Concatenate()([user_flatten, item_flatten, text_input])

    # Neural network layers
    dense = Dense(128, activation='relu',
kernel_regularizer=l2(0.001))(concat)
    dropout = Dropout(0.5)(dense)
    output = Dense(1, activation='linear')(dropout)

    model = Model(inputs=[user_input, item_input, text_input],
outputs=output)
    model.compile(optimizer='adam', loss='mean_squared_error')
    return model

# Initialize and Train the Model
ncf_model = create_ncf_model_with_text(num_users, num_items,
tfidf_matrix.shape[1])
ncf_model.fit([X_train[:, 0], X_train[:, 1], X_train[:, 2:]], y_train,
epochs=5, batch_size=32, verbose=1)

# Evaluate the Model
y_pred_ncf = ncf_model.predict([X_test[:, 0], X_test[:, 1], X_test[:,
2:]]).flatten()
rmse = np.sqrt(mean_squared_error(y_test, y_pred_ncf))

print("NCF Model with Text Feature RMSE:", rmse)
```

ANN Classification

```
import pandas as pd
import numpy as np
import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.metrics import classification_report, roc_auc_score
from tensorflow.keras.callbacks import EarlyStopping

# Load data
data = pd.read_csv('/content/Telco-Customer-Churn.csv')

# Splitting data into features and target
X = data.drop('Churn', axis=1)
y = data['Churn'].apply(lambda x: 1 if x == 'Yes' else 0)

# Identifying numeric and categorical columns
numeric_features = X.select_dtypes(include=['int64',
'float64']).columns
categorical_features = X.select_dtypes(include=['object']).columns

# Creating a Column Transformer for scaling, encoding, and imputing
preprocessor = ColumnTransformer(
    transformers=[
        ('num', Pipeline(steps=[
            ('imputer', SimpleImputer(strategy='mean')),
            ('scaler', StandardScaler())
        ]), numeric_features),
        ('cat', Pipeline(steps=[
            ('imputer', SimpleImputer(strategy='most_frequent')),
            ('onehot', OneHotEncoder(handle_unknown='ignore'))
        ]), categorical_features)])

# Preprocessing the data
X_processed = preprocessor.fit_transform(X).toarray()

# Splitting the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_processed, y,
test_size=0.2, random_state=42)

# Model configuration
model = tf.keras.Sequential([
```

```
    tf.keras.layers.Dense(64, activation='relu',
input_shape=(X_train.shape[1],)),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

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

# Early stopping callback
early_stopping = EarlyStopping(monitor='val_loss', patience=3)

# Model training with early stopping
model.fit(X_train, y_train, epochs=10, batch_size=32,
validation_split=0.2, callbacks=[early_stopping])

# Predicting on test data
y_pred_probs = model.predict(X_test)
y_pred_binary = np.round(y_pred_probs).ravel()

# Model evaluation
accuracy = model.evaluate(X_test, y_test, verbose=0)[1]
print(f"Test Accuracy: {accuracy}")

# Additional Evaluation Metrics
print("\nAdditional Evaluation Metrics:")
print(classification_report(y_test, y_pred_binary))

# AUC-ROC Score
roc_auc = roc_auc_score(y_test, y_pred_probs)
print(f"AUC-ROC Score: {roc_auc}")
```

ANN Regression

```
import pandas as pd
import numpy as np
import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.metrics import mean_squared_error, r2_score
from tensorflow.keras.callbacks import EarlyStopping

# Load data
data = pd.read_csv('/content/Ames_Housing_Data.csv')

# Splitting data into features and target
X = data.drop('SalePrice', axis=1)  # Assuming 'SalePrice' is the
target column
y = data['SalePrice']

# Identifying numeric and categorical columns
numeric_features = X.select_dtypes(include=['int64',
'float64']).columns
categorical_features = X.select_dtypes(include=['object',
'category']).columns

# Creating a Column Transformer for scaling, encoding, and imputing
preprocessor = ColumnTransformer(
    transformers=[
        ('num', Pipeline(steps=[
            ('imputer', SimpleImputer(strategy='mean')),
            ('scaler', StandardScaler())
        ]), numeric_features),
        ('cat', Pipeline(steps=[
            ('imputer', SimpleImputer(strategy='most_frequent')),
            ('onehot', OneHotEncoder(handle_unknown='ignore'))
        ]), categorical_features)])

# Preprocessing the data
X_processed = preprocessor.fit_transform(X).toarray()

# Splitting the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_processed, y,
test_size=0.2, random_state=42)

# Model configuration for regression
```

```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(64, activation='relu',
input_shape=(X_train.shape[1],)),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(1,activation='linear')  # Single output node
for regression
])

model.compile(optimizer='adam',
              loss='mean_squared_error',  # MSE as the loss function
for regression
              metrics=['mean_squared_error'])

# Early stopping callback
early_stopping = EarlyStopping(monitor='val_loss', patience=5)

# Model training with early stopping
model.fit(X_train, y_train, epochs=100, batch_size=32,
validation_split=0.2, callbacks=[early_stopping])

# Predicting on test data
y_pred = model.predict(X_test)

# Model evaluation
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print(f"Test Mean Squared Error: {mse}")
print(f"Test R^2 Score: {r2}")
```

Recommendation System- Content based

```
import pandas as pd
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import re
from nltk.stem import PorterStemmer

# Initialize Stemmer
stemmer = PorterStemmer()

def preprocess_text_without_stopwords(text):
    # Lowercase conversion
    text = text.lower()
    # Removing special characters and digits
    text = re.sub(r'\W+|\d+', ' ', text)
    # Tokenization (splitting text into words)
    words = text.split()
    # Stemming (without stopword removal)
    words = [stemmer.stem(word) for word in words]
    # Joining back to form processed text
    return ' '.join(words)

# Applying preprocessing to overview column without stopwords
data_cleaned['processed_overview'] =
data_cleaned['overview'].apply(preprocess_text_without_stopwords)
```

```
# Text Processing: Processed Overview
tfidf_vectorizer = TfidfVectorizer()
tfidf_matrix =
tfidf_vectorizer.fit_transform(data_cleaned['processed_overview'])

# Similarity Scoring
cosine_sim = cosine_similarity(tfidf_matrix, tfidf_matrix)
```

```
def get_simple_recommendations(title, cosine_sim_matrix, data):
    # Find the index of the movie
    idx = data[data['original_title'] == title].index[0]

    # Get sorted list of tuples (movie_index, similarity_score)
    sorted_movies = sorted(enumerate(cosine_sim_matrix[idx]), key=lambda x:
x[1], reverse=True)

    # Get the titles of the top 10 similar movies
```

```
    top_movies = [data.iloc[i[0]]['original_title'] for i in 
sorted_movies[1:11]]

    return top_movies

# Example usage
simple_recommendations = get_simple_recommendations("Blood and Chocolate", 
cosine_sim, data_cleaned)
print(simple_recommendations)
```

 # text-cleaning/processing

```
sentences = list(df['review'])

corpus = []

for i in range(len(sentences)):
    review = re.sub('[^a-zA-Z]',' ', sentences[i])
    review = review.lower()

    review  = review.split()
    review = [lemmatizer.lemmatize(word) for word in review if not word in 
set(stopwords.words('english')) and len(word)>2]
    review = ' '.join(review)
    corpus.append(review)

cv= CountVectorizer(binary=True, ngram_range=(2,3),max_features=20)
X = cv.fit_transform(corpus)
```

K-Means Clustering

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report,
roc_auc_score
from imblearn.over_sampling import SMOTE
from imblearn.pipeline import Pipeline as ImbPipeline
from sklearn.feature_selection import SelectKBest, f_classif
from sklearn.decomposition import PCA

# Load data
data = pd.read_csv('/content/Telco-Customer-Churn.csv')

# Drop target column for unsupervised learning
X = data.drop('Churn', axis=1)

# Identifying numeric and categorical columns
numeric_features = X.select_dtypes(include=['int64',
'float64']).columns
categorical_features = X.select_dtypes(include=['object']).columns

# Creating a Column Transformer for scaling and encoding
preprocessor = ColumnTransformer(
    transformers=[
        ('num', Pipeline(steps=[
            ('imputer', SimpleImputer(strategy='mean')),
            ('scaler', StandardScaler())
        ]), numeric_features),
        ('cat', Pipeline(steps=[
            ('imputer', SimpleImputer(strategy='most_frequent')),
            ('onehot', OneHotEncoder(handle_unknown='ignore'))
        ]), categorical_features)])

# Preprocessing the data
X_processed = preprocessor.fit_transform(X).toarray()
```

```python
# KMeans Clustering
kmeans = KMeans(n_clusters=3, random_state=42)
clusters = kmeans.fit_predict(X_processed)

# Adding cluster information to the original data
data['Cluster'] = clusters

# Analyzing the Clusters
print(data.groupby('Cluster').mean())

# If needed, visualize the clusters (assuming 2D data, which needs
dimensionality reduction if more than 2 features)
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_processed)
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=clusters, cmap='viridis')
plt.xlabel('PCA Component 1')
plt.ylabel('PCA Component 2')
plt.title('Cluster Visualization')
plt.show()
```

```python
from sklearn.metrics import silhouette_score

silhouette_avg = silhouette_score(X_processed, clusters)
print("Silhouette Score: ", silhouette_avg)
```

word2vec

```
import pandas as pd
import numpy as np
from gensim.models import Word2Vec
from sklearn.metrics.pairwise import cosine_similarity



brand_df = pd.read_csv('/content/brand_category.csv')
category_df = pd.read_csv('/content/categories.csv')
retailer_df = pd.read_csv('/content/offer_retailer.csv')

# Preprocess data
def preprocess_text(text):
    return str(text).lower().split()

brand_df['BRAND'] = brand_df['BRAND'].apply(preprocess_text)
brand_df['BRAND_BELONGS_TO_CATEGORY'] =
brand_df['BRAND_BELONGS_TO_CATEGORY'].apply(preprocess_text)
category_df['PRODUCT_CATEGORY'] =
category_df['PRODUCT_CATEGORY'].apply(preprocess_text)
retailer_df['OFFER'] = retailer_df['OFFER'].apply(preprocess_text)

# Train a Word2Vec model
all_sentences = pd.concat([
    brand_df['BRAND'],
    brand_df['BRAND_BELONGS_TO_CATEGORY'],
    category_df['PRODUCT_CATEGORY'],
    retailer_df['OFFER']
], ignore_index=True)

model = Word2Vec(sentences=all_sentences, vector_size=100, window=5,
min_count=1, workers=4)

# Function to compute vector representation of a text
def get_vector(text):
    vector = np.zeros(100)
    count = 0
    for word in text:
        if word in model.wv:
            vector += model.wv[word]
            count += 1
    if count:
        vector /= count
    return vector

# Get vectors for offers
retailer_df['OFFER_VECTOR'] = retailer_df['OFFER'].apply(get_vector)
```

```python
# Function to get relevant offers
def get_relevant_offers(query):
    query_vector = get_vector(preprocess_text(query))
    similarities = [cosine_similarity([query_vector], [offer_vector])[0][0]
for offer_vector in retailer_df['OFFER_VECTOR']]
    retailer_df['SIMILARITY'] = similarities
    top_offers = retailer_df.sort_values(by='SIMILARITY',
ascending=False).head(3)  # top 3 offers
    return top_offers[['OFFER', 'SIMILARITY']]

print(get_relevant_offers("amazon"))
#print(get_relevant_offers("meat products"))
#print(get_relevant_offers("spend $25"))
```

```python
# Convert labels to numeric format
labels = df['class'].map({'suicide': 1, 'non-suicide': 0})

# Train word2vec model
model = Word2Vec(corpus, sg=1, vector_size=100, window=5, min_count=1,
workers=4)
model.save("/content/gdrive/MyDrive/Colab Data/word2vec.model")

# to train model later
#model = Word2Vec.load("word2vec.model")
#model.train([["hello", "world"]], total_examples=1, epochs=1)(0, 2)

# Cleaning the corpus
X = []
for text in corpus:
    sent_vecs = []
    for word in text:
        if word in model.wv:
            sent_vecs.append(model.wv[word])
    if sent_vecs:
        X.append(np.mean(sent_vecs, axis=0))
    else:
        X.append(np.zeros(model.vector_size))

# Prepare training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, labels, test_size=0.3,
random_state=42)
```

```
# Tensorflow BERT fine-tuning

import tensorflow as tf
from transformers import BertTokenizer, TFBertForSeq2SeqLM
from transformers import Seq2SeqTrainingArguments, Seq2SeqTrainer

# Parameters
BATCH_SIZE = 8
EPOCHS = 3
MAX_LENGTH = 512  # Input length
SUMMARY_LENGTH = 150  # Expected summary length

# Load Pretrained BERT model and Tokenizer
model_name = 'bert-base-uncased'
tokenizer = BertTokenizer.from_pretrained(model_name)
model = TFBertForSeq2SeqLM.from_pretrained(model_name)

# Data Loading & Encoding
def encode_data(original_texts, summaries):
    inputs = tokenizer(original_texts, max_length=MAX_LENGTH, truncation=True, padding='max_length',
return_tensors="tf")
    outputs = tokenizer(summaries, max_length=SUMMARY_LENGTH, truncation=True,
padding='max_length', return_tensors="tf")
    return {
        'input_ids': inputs['input_ids'],
        'attention_mask': inputs['attention_mask']
    }, outputs['input_ids']

# Assuming you have your data in two lists: `original_texts` and `summaries`
train_dataset = tf.data.Dataset.from_tensor_slices((original_texts, summaries))
train_dataset =
train_dataset.map(encode_data).batch(BATCH_SIZE).shuffle(10000).cache().prefetch(buffer_size=tf.data.exp
erimental.AUTOTUNE)
```

```
# Model Compilation & Training
optimizer = tf.keras.optimizers.Adam(learning_rate=3e-5)
loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
model.compile(optimizer=optimizer, loss=loss)

# Train the Model
model.fit(train_dataset, epochs=EPOCHS)

# Summarization
def generate_summary(text):
   encoded_text = tokenizer.encode(text, return_tensors="tf")
   summary_ids = model.generate(encoded_text, max_length=SUMMARY_LENGTH, num_beams=4, length_penalty=2.0, early_stopping=True)
   return tokenizer.decode(summary_ids[0], skip_special_tokens=True)

sample_text = "Your text here for summarization."
print(generate_summary(sample_text))
```

Pytorch Fine-Tuning

```
from transformers import AutoModelForSeq2SeqLM
from peft import get_peft_model, LoraConfig, TaskType
import torch
from datasets import load_dataset
import os

os.environ["TOKENIZERS_PARALLELISM"] = "false"
from transformers import AutoTokenizer
from torch.utils.data import DataLoader
from transformers import default_data_collator, get_linear_schedule_with_warmup
from tqdm import tqdm

device = "cuda"
model_name_or_path = "google/flan-t5-large"
tokenizer_name_or_path = "google/flan-t5-large"

checkpoint_name = "financial_sentiment_analysis_lora_v1.pt"
text_column = "sentence"
label_column = "text_label"
max_length = 128
lr = 1e-7
num_epochs = 3
batch_size = 8


peft_config = LoraConfig(task_type=TaskType.SEQ_2_SEQ_LM, inference_mode=False, r=8, lora_alpha=32,
lora_dropout=0.9)


model = AutoModelForSeq2SeqLM.from_pretrained(model_name_or_path)
model = get_peft_model(model, peft_config)
```

```
model.print_trainable_parameters()



dataset = load_dataset("financial_phrasebank", "sentences_allagree")
dataset = dataset["train"].train_test_split(test_size=0.1)
dataset["validation"] = dataset["test"]
del dataset["test"]

classes = dataset["train"].features["label"].names
dataset = dataset.map(
    lambda x: {"text_label": [classes[label] for label in x["label"]]},
    batched=True,
    num_proc=1,
)

dataset["train"][0]



tokenizer = AutoTokenizer.from_pretrained(model_name_or_path)



def preprocess_function(examples):
    inputs = examples[text_column]
    targets = examples[label_column]
    model_inputs = tokenizer(inputs, max_length=max_length, padding="max_length", truncation=True,
return_tensors="pt")
    labels = tokenizer(targets, max_length=3, padding="max_length", truncation=True, return_tensors="pt")
    labels = labels["input_ids"]
    labels[labels == tokenizer.pad_token_id] = -100
    model_inputs["labels"] = labels
    return model_inputs
```

```
processed_datasets = dataset.map(
    preprocess_function,
    batched=True,
    num_proc=1,
    remove_columns=dataset["train"].column_names,
    load_from_cache_file=False,
    desc="Running tokenizer on dataset",
)

train_dataset = processed_datasets["train"]
eval_dataset = processed_datasets["validation"]

train_dataloader = DataLoader(
    train_dataset, shuffle=True, collate_fn=default_data_collator, batch_size=batch_size, pin_memory=True
)
eval_dataloader = DataLoader(eval_dataset, collate_fn=default_data_collator, batch_size=batch_size,
pin_memory=True)


optimizer = torch.optim.AdamW(model.parameters(), lr=lr)
lr_scheduler = get_linear_schedule_with_warmup(
    optimizer=optimizer,
    num_warmup_steps=0,
    num_training_steps=(len(train_dataloader) * num_epochs),
)


model = model.to(device)

for epoch in range(num_epochs):
    model.train()
    total_loss = 0
    for step, batch in enumerate(tqdm(train_dataloader)):
```

```
        batch = {k: v.to(device) for k, v in batch.items()}
        outputs = model(**batch)
        loss = outputs.loss
        total_loss += loss.detach().float()
        loss.backward()
        optimizer.step()
        lr_scheduler.step()
        optimizer.zero_grad()

    model.eval()
    eval_loss = 0
    eval_preds = []
    for step, batch in enumerate(tqdm(eval_dataloader)):
        batch = {k: v.to(device) for k, v in batch.items()}
        with torch.no_grad():
            outputs = model(**batch)
        loss = outputs.loss
        eval_loss += loss.detach().float()
        eval_preds.extend(
            tokenizer.batch_decode(torch.argmax(outputs.logits, -1).detach().cpu().numpy(),
skip_special_tokens=True)
        )

    eval_epoch_loss = eval_loss / len(eval_dataloader)
    eval_ppl = torch.exp(eval_epoch_loss)
    train_epoch_loss = total_loss / len(train_dataloader)
    train_ppl = torch.exp(train_epoch_loss)
    print(f"{epoch=}: {train_ppl=} {train_epoch_loss=} {eval_ppl=} {eval_epoch_loss=}")


correct = 0
total = 0
for pred, true in zip(eval_preds, dataset["validation"]["text_label"]):
```

```
    if pred.strip() == true.strip():
        correct += 1
    total += 1
accuracy = correct / total * 100
print(f"{accuracy=} % on the evaluation dataset")
print(f"{eval_preds[:10]=}")
print(f"{dataset['validation']['text_label'][:10]=}")


peft_model_id = f"{model_name_or_path}_{peft_config.peft_type}_{peft_config.task_type}"
model.save_pretrained(peft_model_id)
```

Regression using tensorflow in depth

```
import pandas as pd
import numpy as np
import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.feature_selection import SelectKBest, f_regression
from sklearn.pipeline import Pipeline
from sklearn.metrics import mean_squared_error, r2_score
from tensorflow.keras.callbacks import EarlyStopping

# Assuming 'data' is your DataFrame
# Splitting data into features and target
X = data.drop('medicare_patient_hcc_risk_score', axis=1)
y = data['medicare_patient_hcc_risk_score']

# Identifying numeric and categorical columns
numeric_features = X.select_dtypes(include=['int64',
'float64']).columns
categorical_features = X.select_dtypes(include=['object',
'category']).columns

# Creating a Column Transformer for scaling, encoding, and imputing
preprocessor = ColumnTransformer(
    transformers=[
        ('num', Pipeline(steps=[
            ('imputer', SimpleImputer(strategy='mean')),
            ('scaler', StandardScaler())
        ]), numeric_features),
        ('cat', Pipeline(steps=[
            ('imputer', SimpleImputer(strategy='most_frequent')),
            ('onehot', OneHotEncoder(handle_unknown='ignore'))
        ]), categorical_features)])

# Feature Selection
feature_selection = SelectKBest(score_func=f_regression, k='all') #
Adjust 'k' as needed

# Combining preprocessing and feature selection in a pipeline
full_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('feature_selection', feature_selection)
```

```
])

# Preprocessing and feature selection
X_processed = full_pipeline.fit_transform(X,y).toarray()

# Splitting the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_processed, y,
test_size=0.2, random_state=42)

# Update input_shape in model based on the number of selected features
input_shape = X_train.shape[1]

# Model configuration for regression
model = tf.keras.Sequential([
    tf.keras.layers.Dense(64, activation='relu',
input_shape=(input_shape,)),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(1, activation='linear')
])

model.compile(optimizer='adam',
              loss='mean_squared_error',
              metrics=['mean_squared_error'])

# Early stopping callback
early_stopping = EarlyStopping(monitor='val_loss', patience=5)

# Model training with early stopping
history = model.fit(X_train, y_train, epochs=50, batch_size=32,
validation_split=0.2, callbacks=[early_stopping])

plt.plot(history.history['loss'], label='train')
plt.plot(history.history['val_loss'], label='test')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Learning Curve')
plt.legend()
plt.show()

# Predicting on test data
y_pred = model.predict(X_test)

# Model evaluation
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print(f"Test Mean Squared Error: {mse}")
```

```
print(f"Test R^2 Score: {r2}")

from sklearn.metrics import mean_absolute_error

mae = mean_absolute_error(y_test, y_pred)
print(f"Test Mean Absolute Error: {mae}")


from sklearn.metrics import explained_variance_score

explained_variance = explained_variance_score(y_test, y_pred)
print(f"Explained Variance Score: {explained_variance}")

from sklearn.metrics import median_absolute_error

median_ae = median_absolute_error(y_test, y_pred)
print(f"Median Absolute Error: {median_ae}")


import matplotlib.pyplot as plt

residuals = y_test - y_pred.ravel()
plt.scatter(y_test, residuals)
plt.hlines(y=0, xmin=y_test.min(), xmax=y_test.max(), colors='red')
plt.xlabel('Observed Values')
plt.ylabel('Residuals')
plt.title('Residual Plot')
plt.show()
```

ANOVA F-test is a statistical method used to compare the means of two or more groups and determine if they are significantly different from each other. In the context of feature selection in machine learning, the ANOVA F-test is used to select features based on the strength of their relationship with the target variable.

How ANOVA F-test is Used in Feature Selection:

1. **In Regression (`f_regression` in scikit-learn):**
    - `f_regression` computes the correlation between each regressor (feature) and the target variable, and it converts this correlation into an F-score.
    - The F-score captures the degree of linear dependency between each feature and the target. A higher F-score indicates a stronger relationship, implying the feature is more important for prediction.
    - This is particularly effective for linear regression models or when you want to capture linear relationships.
2. **In Classification (`f_classif` in scikit-learn):**

- For classification tasks, `f_classif` is used, which computes the ANOVA F-value between each feature and the target variable (which is categorical).
- It evaluates if the mean of each feature differs significantly across the different classes of the target variable. Again, a higher F-value suggests a feature is more discriminative.

Effectiveness and Limitations:

- **Effectiveness:**
  - For linear models, ANOVA F-test is quite effective as it captures linear dependencies.
  - It's a univariate selection method, meaning each feature is evaluated independently, which makes it computationally efficient.
- **Limitations:**
  - It only captures linear relationships. If the relationship between the feature and target is non-linear, ANOVA F-test may not be able to detect the feature's importance.
  - Being a univariate method, it doesn't account for interactions between features. Features that are useful only in combination with others may not be selected.
  - It can be susceptible to outliers as they can significantly impact mean values and variances.

Conclusion:

In summary, the ANOVA F-test is a quick and effective way to filter features, especially when looking for linear relationships in regression or classification problems. However, its effectiveness can be limited in scenarios involving non-linear relationships or when feature interactions are important. In such cases, other feature selection methods like mutual information, recursive feature elimination, or tree-based feature importances might be more appropriate. The choice of feature selection technique should align with the nature of the data, the problem statement, and the type of model you plan to use.

classification using tensorflow more details

```
import pandas as pd
import numpy as np
import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.feature_selection import SelectKBest,
f_regression,f_classif
from sklearn.pipeline import Pipeline
from sklearn.metrics import mean_squared_error, r2_score
from tensorflow.keras.callbacks import EarlyStopping

# Assuming 'data' is your DataFrame
# Splitting data into features and target
data = pd.read_csv('/content/Telco-Customer-Churn.csv')

data['Churn'] = np.where(data['Churn'] == 'Yes', 1, 0)

X = data.drop('Churn', axis=1)
y = data['Churn']

# Identifying numeric and categorical columns
numeric_features = X.select_dtypes(include=['int64',
'float64']).columns
categorical_features = X.select_dtypes(include=['object',
'category']).columns

# Creating a Column Transformer for scaling, encoding, and imputing
preprocessor = ColumnTransformer(
    transformers=[
        ('num', Pipeline(steps=[
            ('imputer', SimpleImputer(strategy='mean')),
            ('scaler', StandardScaler())
        ]), numeric_features),
        ('cat', Pipeline(steps=[
            ('imputer', SimpleImputer(strategy='most_frequent')),
            ('onehot', OneHotEncoder(handle_unknown='ignore'))
        ]), categorical_features)])

# Feature Selection
feature_selection = SelectKBest(score_func=f_classif, k='all') # Adjust
'k' as needed

# Combining preprocessing and feature selection in a pipeline
full_pipeline = Pipeline(steps=[
```

```
    ('preprocessor', preprocessor),
    ('feature_selection', feature_selection)
])

# Preprocessing and feature selection
X_processed = full_pipeline.fit_transform(X,y).toarray()

# Splitting the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_processed, y,
test_size=0.2, random_state=42)



# Update input_shape in model based on the number of selected features
input_shape = X_train.shape[1]

# Model configuration
model = tf.keras.Sequential([
    tf.keras.layers.Dense(64, activation='relu',
input_shape=(X_train.shape[1],)),
    tf.keras.layers.Dropout(0.1),
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dropout(0.1),
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dropout(0.1),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

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

# Early stopping callback
early_stopping = EarlyStopping(monitor='val_loss', patience=3)

# Model training with early stopping
history = model.fit(X_train, y_train, epochs=10, batch_size=32,
validation_split=0.2, callbacks=[early_stopping])

# Predicting on test data
y_pred_probs = model.predict(X_test)
y_pred_binary = np.round(y_pred_probs).ravel()

# Model evaluation
accuracy = model.evaluate(X_test, y_test, verbose=0)[1]
print(f"Test Accuracy: {accuracy}")

# Additional Evaluation Metrics
print("\nAdditional Evaluation Metrics:")
```

```python
print(classification_report(y_test, y_pred_binary))

# AUC-ROC Score
roc_auc = roc_auc_score(y_test, y_pred_probs)
print(f"AUC-ROC Score: {roc_auc}")
plt.plot(history.history['loss'], label='train')
plt.plot(history.history['val_loss'], label='test')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Learning Curve')
plt.legend()
plt.show()

# Predicting on test data
y_pred = model.predict(X_test)
y_pred = (y_pred > 0.5).astype(int)  # Converting probabilities to 
binary predictions

# Assuming y_test is binary (0s and 1s)
from sklearn.metrics import roc_auc_score

# ROC-AUC Score
roc_auc = roc_auc_score(y_test, y_pred)
print(f'ROC-AUC Score: {roc_auc}')
```