Loading the packages

```
import random
import re
import pandas as pd
import numpy as np
from numpy import linalg as LA
from sklearn.decomposition import PCA
from sklearn.feature_selection import SelectKBest, f_regression
from sklearn.linear_model import LinearRegression as SklearnLinearRegression
from sklearn.linear_model import LogisticRegression as SklearnLogisticRegression
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from datetime import datetime
from sklearn.model_selection import train_test_split, KFold
from sklearn.metrics import mean_squared_error, classification_report, accuracy_score, roc_auc_score
```

Loading the datasets

```
test_data = pd.read_csv("/content/test.csv")
train_data = pd.read_csv("/content/train.csv")
```

Class of PCA

```
class CustomPCA:
    """
    Principal component analysis (PCA):
        1. Standardize the continuous initial variables
        2.Compute the covariance matrix
        3.Compute the eigenvectors and eigenvalues of the covariance matrix
        4.Select the top-k eigenvectors by their corresponding eigenvalues
        5.Transform the original data along the axes of the principal component
    """

    def __init__(self, n_components):
        self.n_components = n_components
        self.eigenvalues = None
        self.eigenvectors = None

    def fit(self, X):
        # Standardize the data
        X_std = (X - np.mean(X, axis=0)) / np.std(X, axis=0, ddof=1)

        # Calculate the covariance matrix
        cov_mat = np.cov(X_std, rowvar=False, bias=False)

        # Perform eigen decomposition
        w, v = LA.eig(cov_mat)

        # Sort the eigenvalues and eigenvectors in decreasing order
        idx = w.argsort()[::-1]
        w = w[idx]
        v = v[:, idx]

        # Select the top n_components eigenvalues and eigenvectors
        self.eigenvalues = w[:self.n_components]
        self.eigenvectors = v[:, :self.n_components].T

    def transform(self, X):
        # Standardize the data
        X_std = (X - np.mean(X, axis=0)) / np.std(X, axis=0, ddof=1)

        # Project the data onto the principal components
        return np.matmul(X_std, self.eigenvectors)

    def fit_transform(self, X):
        self.fit(X)
        return self.transform(X)
```

Class of Linear Regression Model

```
class LinearRegressionModel:
    def __init__(self, learning_rate=0.04, n_iterations=3000):
        self.learning_rate = learning_rate
        self.n_iterations = n_iterations
        self.weights, self.bias = None, None
```

```
    def fit(self, X, y):
        n_samples, n_features = X.shape

        # Initialize weights and bias
        self.weights = np.random.randn(n_features)
        self.bias = 0

        # Gradient Descent
        for _ in range(self.n_iterations):
            y_predicted = np.dot(X, self.weights) + self.bias

            # Compute gradients
            dw = (1 / n_samples) * np.dot(X.T, (y_predicted - y))
            db = (1 / n_samples) * np.sum(y_predicted - y)

            # Update parameters
            self.weights -= self.learning_rate * dw
            self.bias -= self.learning_rate * db

    def predict(self, X):
        return np.dot(X, self.weights) + self.bias
```

class of Logistic Regression Model

```
class LogisticRegressionModel:
    """
    A class which implements logistic regression model with gradient descent.
    """

    def __init__(self, learning_rate=0.1, n_iterations=3000):
        self.learning_rate = learning_rate
        self.n_iterations = n_iterations
        self.weights, self.bias = None, None

    @staticmethod
    def _sigmoid(x):
        """
        Private method, used to pass results of the line equation through the sigmoid function.

        :param x: float, prediction made by the line equation
        :return: float
        """
        return 1 / (1 + np.exp(-x))

    @staticmethod
    def _binary_cross_entropy(y, y_hat):
        '''
        Private method, used to calculate binary cross entropy value between actual classes
        and predicted probabilities.

        :param y: array, true class labels
        :param y_hat: array, predicted probabilities
        :return: float
        '''

        def safe_log(x):
            return 0 if x == 0 else np.log(x)

        total = 0
        for curr_y, curr_y_hat in zip(y, y_hat):
            total += (curr_y * safe_log(curr_y_hat) + (1 - curr_y) * safe_log(1 - curr_y_hat))
        return - total / len(y)

    def fit(self, X, y):
        '''
        Used to calculate the coefficient of the logistic regression model.

        :param X: array, features
        :param y: array, true values
        :return: None
        '''
        # 1. Initialize coefficients
        self.weights = np.zeros(X.shape[1])
        self.bias = 0

        # 2. Perform gradient descent
        for i in range(self.n_iterations):
            linear_pred = np.dot(X, self.weights) + self.bias
            probability = self._sigmoid(linear_pred)
```

```
            # Calculate derivatives
            partial_w = (1 / X.shape[0]) * (np.dot(X.T, (probability - y)))
            partial_d = (1 / X.shape[0]) * (np.sum(probability - y))

            # Update the coefficients
            self.weights -= self.learning_rate * partial_w
            self.bias -= self.learning_rate * partial_d

    def predict_proba(self, X):
        '''
        Calculates prediction probabilities for a given threshold using the line equation
        passed through the sigmoid function.

        :param X: array, features
        :return: array, prediction probabilities
        '''
        linear_pred = np.dot(X, self.weights) + self.bias
        return self._sigmoid(linear_pred)

    def predict(self, X, threshold=0.5):
        '''
        Makes predictions using the line equation passed through the sigmoid function.

        :param X: array, features
        :param threshold: float, classification threshold
        :return: array, predictions
        '''
        probabilities = self.predict_proba(X)
        return [1 if i > threshold else 0 for i in probabilities]
```

Preproccesion of the data: fill na, stantartization and etc.

```
def preprocess_data(data):
    # Drop specified columns
    data.drop(['id', 'host_id', 'property_type', 'has_availability', 'first_review', 'last_review', 'license'], axis=1,
              inplace=True)
    # Fill missing values:
    response_time_to_score = {"within an hour": 4, "within a few hours": 3, "within a day": 2,
                              "a few days or more": 1, }

    def map_score(response_time):
        if pd.isnull(response_time) or response_time.strip() == '':
            return 2.5
        else:
            return response_time_to_score.get(response_time, 2.5)

    data['host_response_time_score'] = data['host_response_time'].apply(map_score)

    data['host_response_rate'] = data['host_response_rate'].str.rstrip('%').astype('float')
    # Fill NA/NaN values with the median
    median_val = data['host_response_rate'].median()
    data['host_response_rate'] = data['host_response_rate'].fillna(median_val)

    data['host_acceptance_rate'] = data['host_acceptance_rate'].str.rstrip('%').astype('float')
    # Fill NA/NaN values with the median
    median_val = data['host_acceptance_rate'].median()
    data['host_acceptance_rate'] = data['host_acceptance_rate'].fillna(median_val)

    data['host_is_superhost'].fillna(data['host_is_superhost'].mode()[0], inplace=True)
    data['host_has_profile_pic'].fillna(data['host_has_profile_pic'].mode()[0], inplace=True)
    data['host_identity_verified'].fillna('f', inplace=True)

    def map_bathrooms_text_score(bathrooms_text):
        if pd.isnull(bathrooms_text) or bathrooms_text.strip() == '':
            return 1.5  # default value for NA or empty string

        # Extract the number of baths
        number = re.findall("\d+\.\d+|\d+", bathrooms_text)
        if number:
            number_score = float(number[0])
        else:
            number_score = 0  # default value for no number information

        # Check the type of baths
        if "shared" in bathrooms_text:
            type_score = 1
        elif "private" in bathrooms_text:
            type_score = 2
        else:
```

```
        type_score = 1.5  # default value for no type information

    return number_score * type_score

data['bathrooms_text_score'] = data['bathrooms_text'].apply(map_bathrooms_text_score)
data.drop('bathrooms_text', axis=1, inplace=True)

data['host_listings_count'].fillna(data['host_listings_count'].median(), inplace=True)
data['latitude'].fillna(data['latitude'].median(), inplace=True)
data['longitude'].fillna(data['longitude'].median(), inplace=True)
data['accommodates'].fillna(data['accommodates'].median(), inplace=True)

data['availability_30'].fillna(data['availability_30'].median(), inplace=True)
data['availability_60'].fillna(data['availability_60'].median(), inplace=True)
data['availability_90'].fillna(data['availability_90'].median(), inplace=True)
data['availability_365'].fillna(data['availability_365'].median(), inplace=True)

data['number_of_reviews'].fillna(data['number_of_reviews'].median(), inplace=True)
data['number_of_reviews_ltm'].fillna(data['number_of_reviews_ltm'].median(), inplace=True)
data['number_of_reviews_l30d'].fillna(data['number_of_reviews_l30d'].median(), inplace=True)

data['review_scores_rating'].fillna(data['review_scores_rating'].median(), inplace=True)
data['review_scores_accuracy'].fillna(data['review_scores_accuracy'].median(), inplace=True)
data['review_scores_cleanliness'].fillna(data['review_scores_cleanliness'].median(), inplace=True)
data['review_scores_checkin'].fillna(data['review_scores_checkin'].median(), inplace=True)
data['review_scores_communication'].fillna(data['review_scores_communication'].median(), inplace=True)
data['review_scores_location'].fillna(data['review_scores_location'].median(), inplace=True)
data['review_scores_value'].fillna(data['review_scores_value'].median(), inplace=True)
data['calculated_host_listings_count_entire_homes'].fillna(
    data['calculated_host_listings_count_entire_homes'].median(), inplace=True)
data['calculated_host_listings_count_private_rooms'].fillna(
    data['calculated_host_listings_count_private_rooms'].median(), inplace=True)
data['calculated_host_listings_count_shared_rooms'].fillna(
    data['calculated_host_listings_count_shared_rooms'].median(), inplace=True)

data['instant_bookable'].fillna(data['instant_bookable'].mode()[0], inplace=True)

data['host_total_listings_count'].fillna(data['host_total_listings_count'].median(), inplace=True)
data['bedrooms'].fillna(0, inplace=True)
data['beds'].fillna(0, inplace=True)
data['calculated_host_listings_count'].fillna(0, inplace=True)
data['reviews_per_month'].fillna(0, inplace=True)
review_columns = ['review_scores_rating', 'review_scores_accuracy', 'review_scores_cleanliness',
                  'review_scores_checkin', 'review_scores_communication',
                  'review_scores_location', 'review_scores_value', 'reviews_per_month']
for column in review_columns:
    if data[column].dtype == 'object':
        data[column] = data[column].fillna('No Review')
    else:
        data[column].fillna(0, inplace=True)

# Fill remaining missing values
nights_max = ['maximum_nights', 'maximum_minimum_nights', 'maximum_maximum_nights', 'maximum_nights_avg_ntm']
for column in nights_max:
    data[column].fillna(data[column].median(), inplace=True)

nights_min = ['minimum_nights', 'minimum_minimum_nights', 'minimum_maximum_nights', 'minimum_nights_avg_ntm']
for column in nights_min:
    data[column].fillna(0, inplace=True)

# Identify columns with 't' and 'f' values
boolean_columns = [col for col in data.columns if set(data[col].unique()) == {'f', 't'}]

# Convert 't' and 'f' values to binary
for col in boolean_columns:
    data[col] = data[col].map({'f': 0, 't': 1})

# Convert to datetime format and calculate the number of days from the date to now
date_columns = ['host_since']
for col in date_columns:
    data[col] = pd.to_datetime(data[col], errors='coerce')
    data[col] = (datetime.now() - data[col]).dt.days
    data[col] = data[col].fillna(0.5)

# Apply MinMaxScaler to date columns
scaler = MinMaxScaler()
data[date_columns] = scaler.fit_transform(data[date_columns])

# Convert 'amenities' to the sum of the list
data['amenities'] = data['amenities'].fillna('[]').apply(lambda x: len(eval(x)))

room_type_to_score = {'Entire home/apt': 3, 'Private room': 2, 'Shared room': 1, 'Hotel room': 2, }
```

```
    def map_room_type_score(room_type):
        if pd.isnull(room_type) or room_type.strip() == '':
            return 0.5  # or any default value you'd like to assign
        else:
            return room_type_to_score.get(room_type, 2)

    data['room_type'] = data['room_type'].apply(map_room_type_score)
    data.drop('room_type', axis=1, inplace=True)

    # Extract three unique values from 'host_verifications'
    data['host_verifications'] = data['host_verifications'].fillna('[]')
    unique_verifications = data['host_verifications'].apply(eval).explode().unique()[:3]

    # Create dummy variables for 'host_verifications' using the unique values
    host_verifications_dummies = pd.DataFrame()
    for verification in unique_verifications:
        column_name = 'host_verification_' + verification
        host_verifications_dummies[column_name] = data['host_verifications'].apply(
            lambda row: verification in row).astype(int)

    # Concatenate the dummy variables at the beginning of the DataFrame
    data = pd.concat([host_verifications_dummies, data], axis=1)
    data.drop('host_verifications', axis=1, inplace=True)

    data = data.select_dtypes(exclude=['object'])

    # Detect anomalies in the numerical columns
    numerical_columns = data.select_dtypes(include=['int64', 'float64']).columns
    anomalies = {}
    for column in numerical_columns:
        mean = data[column].mean()
        std = data[column].std()
        lower_bound = mean - 3 * std
        upper_bound = mean + 3 * std
        anomalies[column] = data[(data[column] < lower_bound) | (data[column] > upper_bound)]

    # Apply StandardScaler to each column (excluding 'expensive')
    scaler = StandardScaler()
    data_scaled = data.copy()  # Create a copy of the original DataFrame
    columns_to_scale = [col for col in data.columns if col != 'expensive']
    data_scaled[columns_to_scale] = scaler.fit_transform(data[columns_to_scale])
    return data_scaled
```

Cross Validation:

```
def cross_val_PCA_selections_Logistic(train_data, k_pca, k_selection, cv=10, seed=2023):
    """
    Performs cross-validation on the Logistic Regression model with PCA and SelectKBest feature selection.

    Parameters:
    -----------
    train_data : pandas.DataFrame
        The training dataset with features and target variable.

    k_pca : int
        The number of principal components to retain during PCA.

    k_selection : int
        The number of best features to retain during feature selection with SelectKBest.

    cv : int, default=10
        The number of folds for cross-validation.

    seed : int, default=2023
        The random seed to ensure reproducibility of results.

    Returns:
    --------
    None. Prints out the AUC-ROC scores from cross-validation, their mean and standard deviation.
    """

    # Preprocess and scale the data
    train_data_scaled = preprocess_data(train_data)
    X_train = train_data_scaled.drop(columns=['expensive'])
    y_train = train_data_scaled['expensive']

    # Apply PCA to reduce dimensionality
    pca = CustomPCA(n_components=k_pca)
    X_train_pca = pca.fit_transform(X_train)
```

```
    # Apply SelectKBest to select most informative features
    k_best = SelectKBest(score_func=f_regression, k=k_selection)
    X_train_selected = k_best.fit_transform(X_train_pca, y_train)

    # Initialize logistic regression model
    custom_lr = LogisticRegressionModel()

    # Perform cross-validation and calculate AUC-ROC scores
    kf = KFold(n_splits=cv, random_state=seed, shuffle=True)
    auc_scores = []

    # Iterate through each fold
    for train_index, val_index in kf.split(X_train_selected):
        # Split the data into training and validation sets for this fold
        X_train_fold, X_val_fold = X_train_selected[train_index], X_train_selected[val_index]
        y_train_fold, y_val_fold = y_train[train_index], y_train[val_index]

        # Fit the model on the training set
        custom_lr.fit(X_train_fold, y_train_fold)

        # Make predictions on the validation set
        y_val_pred = custom_lr.predict(X_val_fold)

        # Calculate the AUC-ROC score for these predictions
        auc_score = roc_auc_score(y_val_fold, y_val_pred)
        auc_scores.append(auc_score)

    # Convert list to numpy array for convenience
    auc_scores = np.array(auc_scores)

    # Print the results
    print("Cross-Validation Scores:", auc_scores)
    print("Mean AUC-ROC:", auc_scores.mean())
    print("Standard Deviation:", auc_scores.std())
```

Comparing our model implementaions to sklearn:

The PCA and Linear regression results are the same. The Logistic Regression models are silightly different since there is an omptimizations that the sklearn do to get more accurate probabilities.

```
def comparing_VS_Sklearn(train_data):
    print( "--------------- Logistic Regression Model -----------------")

    # Prepare the data
    X_Logistic = train_data[['reviews_per_month', 'calculated_host_listings_count', 'review_scores_rating', 'number_of_reviews']].fillna
    y = train_data["expensive"]

    # Scale the features
    scaler = StandardScaler()
    X_Logistic = scaler.fit_transform(X_Logistic)

    # Split dataset into train and test sets
    X_train, X_test, y_train, y_test = train_test_split(X_Logistic, y, test_size=0.2, random_state=2023)

    # Create an instance of the Logistic Regression class and fit the model
    model_custom = LogisticRegressionModel()
    model_custom.fit(X_train, y_train)

    # Predict the probabilities
    my_predictions = model_custom.predict_proba(X_test)

    # Print the model predictions for class 1
    print("Custom Model Predictions:\n", my_predictions[:8])

    # Create an instance of the Logistic Regression class from sklearn and fit the model
    model_sklearn = SklearnLogisticRegression()
    model_sklearn.fit(X_train, y_train)

    # Predict the probabilities
    sklearn_predictions = model_sklearn.predict_proba(X_test)

    # Print the sklearn model predictions for class 1
    print("\nSklearn Model Predictions:\n", sklearn_predictions[:8, 1])

    print( "\n --------------- PCA -----------------")

    # Prepare the data
    X_PCA = train_data[['reviews_per_month', 'calculated_host_listings_count', 'review_scores_rating', 'number_of_reviews']].fillna(0)

    # Standardize the data
    X_std = (X_PCA - np.mean(X_PCA, axis=0)) / np.std(X_PCA, axis=0, ddof=1)
```

```
    # Fit and print results for your PCA model
    my_model = CustomPCA(n_components=3)
    my_model.fit(X_std.values)
    print("My model selected vectors:")
    print(my_model.eigenvectors)

    # Fit and print results for sklearn PCA model
    sk_model = PCA(n_components=3)
    sk_model.fit(X_std.values)
    print("\n Scikit-learn model selected vectors:")
    print(sk_model.components_)

    print( "\n --------------- Linear Regression -----------------")

    X_linearReg = train_data[['reviews_per_month', 'calculated_host_listings_count', 'review_scores_rating', 'number_of_reviews']].filln

    # Standardize the features to have mean=0 and variance=1
    scaler = StandardScaler()
    X_linearReg = scaler.fit_transform(X_linearReg)

    y = train_data["expensive"]

    # Your Linear Regression Model
    print("Running Custom Linear Regression Model...")
    my_model = LinearRegressionModel()
    my_model.fit(X_linearReg, y)
    print("My Model Coefficients:\n", my_model.weights)

    # Sklearn's Linear Regression Model
    print("\n Running Sklearn's Linear Regression Model...")
    sklearn_model = SklearnLinearRegression()
    sklearn_model.fit(X_linearReg, y)
    print("\n Sklearn's Model Coefficients:\n", sklearn_model.coef_)

comparing_VS_Sklearn(train_data)
```

```
 --------------- Logistic Regression Model -----------------
Custom Model Predictions:
 [0.67995807 0.80418235 0.70340957 0.82080262 0.60497382 0.66471289
 0.69264192 0.69694955]

Sklearn Model Predictions:
 [0.67982472 0.8047032  0.70331428 0.82138169 0.6051306  0.66513191
 0.69257384 0.69685377]

 --------------- PCA -----------------
My model selected vectors:
[[-0.63792096  0.24666573 -0.35628703 -0.6366101 ]
 [ 0.28394195  0.81080352 -0.43344351  0.27221564]
 [-0.1032672   0.53018823  0.8273354  -0.15411836]]

 Scikit-learn model selected vectors:
[[ 0.63792096 -0.24666573  0.35628703  0.6366101 ]
 [ 0.28394195  0.81080352 -0.43344351  0.27221564]
 [ 0.1032672  -0.53018823 -0.8273354   0.15411836]]

 --------------- Linear Regression -----------------
Running Custom Linear Regression Model...
My Model Coefficients:
 [ 0.12859772 -0.04312345 -0.00444707 -0.09252031]

 Running Sklearn's Linear Regression Model...

 Sklearn's Model Coefficients:
 [ 0.12859773 -0.04312345 -0.00444707 -0.09252031]
```

The final code which creates a csv of id and its probability:

```
def FinalSubmission(train_data):
    # Load test dataset
    id = test_data['id']
    train_data_scaled = preprocess_data(train_data)

    X_train = train_data_scaled.drop(columns=['expensive'])
    y_train = train_data_scaled['expensive']

    X_test = preprocess_data(test_data)

    # Use your custom Logistic Regression class
    custom_lr = LogisticRegressionModel(learning_rate=0.1, n_iterations=1000)
    custom_lr.fit(X_train.values, y_train.values)
    custom_probabilities = custom_lr.predict_proba(X_test.values)
```

```
        return id, custom_probabilities

    id, prob = FinalSubmission(train_data)
    predictions_df = pd.DataFrame({'id': id, 'expensive': prob})
    predictions_df.to_csv('predicted_probabilities.csv', index=False)
    predictions_df
```

```
<ipython-input-118-3de26dbd9457>:118: UserWarning: Parsing dates in DD/MM/YYYY forma
  data[col] = pd.to_datetime(data[col], errors='coerce')
<ipython-input-118-3de26dbd9457>:118: UserWarning: Parsing dates in DD/MM/YYYY forma
  data[col] = pd.to_datetime(data[col], errors='coerce')
```

	id	expensive
0	6760	0.643171
1	6761	0.936023
2	6762	0.876789
3	6763	0.934544
4	6764	0.877105
...	...	...
3022	9782	0.962575
3023	9783	0.993449
3024	9784	0.992463
3025	9785	0.682046
3026	9786	0.743131

3027 rows × 2 columns

✓ 2s completed at 11:46 PM