DATA 201 - Assignment 2

Please use this page http://apps.ecs.vuw.ac.nz/submit/DATA201 (http://apps.ecs.vuw.ac.nz/submit/DATA201) for submitssion and submit only this single Jupyter notebook with your code added into it at the appropriate places.

The due date is **Sunday 5th April, before midnight**.

The dataset for this assignment is file **sales_data.csv** which is provided with this notebook.

Please choose menu items *Kernel => Restart & Run All* then *File => Save and Checkpoint* in Jupyter before submission.

Problem Statement

A retail company wants to understand the customer purchase behaviour (specifically, purchase amount) against various products of different categories. They have shared purchase summary of various customers for selected high volume products from last month. The data set also contains customer demographics (age, gender, marital status, city_type, stay_in_current_city), product details (product_id and product category) and total purchase_amount from last month.

You need to build a model to predict the purchase amount of customer against various products which will help the company to create personalized offer for customers against different products.

Data

Variable	Description
`User_ID`	User ID
`Product_ID`	Product ID
`Gender`	Sex of User
`Age`	Age in bins
`Occupation`	Occupation (Masked)
`City_Category`	Category of the City (A, B, C)
`Stay_In_Current_City_Years`	Number of years stay in current city
`Marital_Status`	Marital Status
`Product_Category_1`	Product Category (Masked)
`Product_Category_2`	Product may belongs to other category also (Masked)
`Product_Category_3`	Product may belongs to other category also (Masked)
`Purchase`	Purchase Amount (Target Variable)

Evaluation

The root mean squared error (RMSE) will be used for model evaluation.

Questions and Code

In [1]:

```
import numpy as np
import pandas as pd
from sklearn import metrics
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsRegressor
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder, MinMaxScaler, StandardScaler

np.random.seed = 42
```

Load the given dataset.

In [2]:

```
data = pd.read_csv("sales_data.csv")
data.dtypes
```

Out[2]:

```
Age                          object
City_Category                object
Gender                       object
Marital_Status                int64
Occupation                    int64
Product_Category_1            int64
Product_Category_2            int64
Product_Category_3            int64
Product_ID                    int64
Purchase                    float64
Stay_In_Current_City_Years   object
User_ID                       int64
dtype: object
```

1. Is there any missing value? [1 point] 1/1

In [3]:

```
# There are no missing values
data.isna().sum()
```

Out[3]:

```
Age                          0
City_Category                0
Gender                       0
Marital_Status               0
Occupation                   0
Product_Category_1           0
Product_Category_2           0
Product_Category_3           0
Product_ID                   0
Purchase                     0
Stay_In_Current_City_Years   0
User_ID                      0
dtype: int64
```

2. Drop attribute `User_ID` . [1 point]

In [4]:

1/1

```
data = data.drop('User_ID', axis=1)
```

3. Then convert the following categorical attributes below to numerical values with the rule as below. [4 points] 4/4

- `Gender` : `F` :0, `M` :1
- `Age` : `0-17` :0, `18-25` :1, `26-35` :2, `36-45` :3, `46-50` :4, `51-55` :5, `55+` :6
- `Stay_In_Current_City_Years` : `0` :0, `1` :1, `2` :2, `3` :3, `4+` :4

You may want to apply a `lambda` function to each row of a column in the dataframe. Some examples here may be helpful: https://thispointer.com/pandas-apply-apply-a-function-to-each-row-column-in-dataframe/ (https://thispointer.com/pandas-apply-apply-a-function-to-each-row-column-in-dataframe/)

In [5]:

```
data['Gender'] = data['Gender'].map({'F':0,'M':1})
data['Age'] = data['Age'].map({'0-17':0,'18-25':1, '26-35':2, '36-45':3, '46-50':4, '51
-55':5, '55+':6})
data['Stay_In_Current_City_Years'] = data['Stay_In_Current_City_Years'].map({'0':0,'1':
1, '2':2, '3':3, '4+':4})
data.head()
```

Out[5]:

	Age	City_Category	Gender	Marital_Status	Occupation	Product_Category_1	Product_Ca
0	0	A	0	0	10	1	
1	4	B	1	1	7	1	
2	2	A	1	1	20	1	
3	5	A	0	0	9	5	
4	5	A	0	0	9	2	

4. Randomly split the current data frame into 2 subsets for training (80%) and test (20%). Use *random_state = 42*. [2 points]

2/2

In [6]:

```
data_train, data_test = train_test_split(data, test_size=0.2, random_state=42)
```

5. Get the list of numerical predictors (all the attributes in the current data frame except the target, `Purchase`) and the list of categorical predictor. [1 point]

This question asked for the lists of numerical and categorical variables:

In [7]: num_cols and nom_cols as produced later, correctly, so 1/1

```
X_train = data_train.drop('Purchase', axis=1)
y_train = data_train['Purchase'].copy()

X_test = data_test.drop('Purchase', axis=1)
y_test = data_test['Purchase'].copy()
```

6. Create a transformation pipeline including two pipelines handling the following [3 points]

- Numerical *predictors*: apply Standard Scaling
- Categorical *predictor*: apply One-hot-encoding

3/3

You will need to use `ColumnTransformer` . The example in Week 3 lectures may be helpful.

In [8]:

```
nom_onehot = [('onehot', OneHotEncoder(sparse=False, handle_unknown='ignore'))]
nom_pl = Pipeline(nom_onehot)

num_impute = SimpleImputer(strategy='mean')
num_normalised = MinMaxScaler()
num_pl = Pipeline([('imp', num_impute), ('norm', num_normalised)])

num_cols = list(X_train.select_dtypes([np.number]).columns)
nom_cols = list(set(X_train.columns) - set(num_cols))

transformers = [('num', num_pl, num_cols),
                ('nom', nom_pl, nom_cols)]

col_transform = ColumnTransformer(transformers)
```

7. Train and use that transformation pipeline to transform the training data (e.g. for a machine learning model). [2 points] 2/2

In [9]:

```
X_train_trans = col_transform.fit_transform(X_train)
X_train_trans
```

Out[9]:

```
array([[0.33333333, 0.        , 1.        , ..., 1.        , 0.        ,
        0.        ],
       [0.33333333, 1.        , 0.        , ..., 1.        , 0.        ,
        0.        ],
       [0.33333333, 1.        , 0.        , ..., 0.        , 1.        ,
        0.        ],
       ...,
       [0.16666667, 1.        , 0.        , ..., 0.        , 1.        ,
        0.        ],
       [0.5       , 1.        , 0.        , ..., 1.        , 0.        ,
        0.        ],
       [0.66666667, 1.        , 1.        , ..., 0.        , 1.        ,
        0.        ]])
```

8. Use that transformation pipeline to transform the test data (e.g. for testing a machine learning model). [2 points]

2/2

In [10]:

```
X_test_trans = col_transform.transform(X_test)
X_test_trans
```

Out[10]:

```
array([[0.5       , 1.        , 1.        , ..., 1.        , 0.        ,
        0.        ],
       [0.16666667, 1.        , 1.        , ..., 0.        , 1.        ,
        0.        ],
       [0.5       , 0.        , 1.        , ..., 0.        , 0.        ,
        1.        ],
       ...,
       [0.33333333, 0.        , 0.        , ..., 0.        , 0.        ,
        1.        ],
       [0.        , 0.        , 0.        , ..., 1.        , 0.        ,
        0.        ],
       [0.5       , 1.        , 0.        , ..., 0.        , 1.        ,
        0.        ]])
```

9. Build a Linear Regression model using the training data after transformation and test it on the test data. Report the RMSE values on the training and test data. [3 points] 3/3

Document: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html)

In [11]:

```
lr = LinearRegression()
lr_pipeline = Pipeline([('col_trans', col_transform), ('lr', lr)])
lr_pipeline.fit(X_train, y_train)

lr_train_pred = lr_pipeline.predict(X_train)
lr_test_pred = lr_pipeline.predict(X_test)
print(" Linear Regression Training Set RMSE: %.4g" % np.sqrt(metrics.mean_squared_error
(lr_train_pred, y_train)))
print("Linear Regression Test Set RMSE: %.4g" % np.sqrt(metrics.mean_squared_error(lr_t
est_pred, y_test)))
```

```
 Linear Regression Training Set RMSE: 4600
Linear Regression Test Set RMSE: 4616
```

10. Repeat Question 9 using a `KNeighborsRegressor` . Comment on the processing time and performance of the model in this question. [1 point]

Document: https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsRegressor.html (https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsRegressor.html)

In [12]:

```
knn = KNeighborsRegressor()
knn_pipeline = Pipeline([('col_trans', col_transform), ('knn', knn)])
knn_pipeline.fit(X_train, y_train)

knn_train_pred  = knn_pipeline.predict(X_train)
knn_test_pred = knn_pipeline.predict(X_test)
print("K Neighbours Regressor Training Set RMSE: %.4g" % np.sqrt(metrics.mean_squared_e
rror(knn_train_pred, y_train)))
print("K Neighbours Regressor Test Set RMSE: %.4g" % np.sqrt(metrics.mean_squared_error
(knn_test_pred, y_test)))
```

```
K Neighbours Regressor Training Set RMSE: 3407
K Neighbours Regressor Test Set RMSE: 4230
```

The K-Nearest Neighbours Regression is significantly slower than Linear Regression because in KNN each training instance has to be compared with every other training instance one-by-one, this makes it computationally expensive especially when your dataset is wide (a lot of features, which in our instance, it does). It's complexity is n $\times$ n because each instance has n comparisions.

KNN Regression also has poorer performance than the Linear Regression model, as observed by how RMSE for the training and test sets for Linear Regression are near identicial, it means there's little variance in the residuals. KNN Regression on the other hand has a pretty big discrepancy between the RMSE values, most likely due to a very terrible signal-to-noise ratio thanks to our large number of features. KNN works best when it comes to small datasets without too much noise

In [ ]: