databricks ACT17

(https://databricks.com)

```
# File location and type
file_location = "/FileStore/tables/chi_sq2.csv"
file_type = "csv"

# CSV options
infer_schema = "true"
first_row_is_header = "true"
delimiter = ","

# The applied options are for CSV files. For other file types, these will be ignored.
df = spark.read.format(file_type) \
  .option("inferSchema", infer_schema) \
  .option("header", first_row_is_header) \
  .option("sep", delimiter) \
  .load(file_location)

display(df)
```

Table

New result table: ON

	marital	housing	label
1	married	no	0
2	single	yes	0
3	single	no	1
4	single	yes	0
5	single	no	0
6	single	yes	0
7	single	no	1
8	single	yes	1
9	single	no	0
10	married	yes	0
11	married	no	1
12	married	yes	0
13	married	yes	0
14	married	yes	0
15	married	yes	0

51 rows

```
df.printSchema()
```

```
root
 |-- marital: string (nullable = true)
 |-- housing: string (nullable = true)
 |-- label: integer (nullable = true)
```

```
df.count()
```

```
Out[3]: 51
```

our dataframe has 51 rows

```
from pyspark.ml.feature import StringIndexer
```

I have imported StringIndexer from pyspark.

```
#Create a StringIndexer to convert categorical variables to numerical values
marital_indexer = StringIndexer(inputCol="marital",outputCol="marital_num").fit(df)
```

The avobe line of code is a powerfull tool for handling large-scale data procesing and analysis. This line of code involves the StringIndexer class and we are going to use it for feature transportation
StingIndexer is a class from PySpark's pyspark.ml.feature module.
We are using StringIndexer to encode a string column of labels to a columns of label indices, that we are going to transform from non-numerical labels to a numerical format.
We are using the marital column as our inputCol
the .fit method is applied to the StringIndexer instance so that it learns the mapping from the labels in the marital column to the numeric indices.
The object marital_indexer is using the marital columns to create a new column marital_num with numerical indices

```
df1 = marital_indexer.transform(df)
```

Here we are applying the the .transform method to the marital_indexer. We are taking the df dataframe and creating a new df1 dataframe with the string-to-index mapping learned by marital_indexer.
Remember df1 has a new column marital_num that can be use in ML workflows where categorical string data is converted to numeric form so it can be used in various machine learning algorithms.

I applied a StringIndexer called marital_indexer on df1 to transform the marital column to marital_num.

```
from pyspark.ml.feature import OneHotEncoder
```

I imported OneHotEncoder from pyspark.

```
# Create a OneHotEncoder to convert numerical values to binary vectors
marital_encoder = OneHotEncoder(inputCol="marital_num", outputCol="marital_vector")
marital_encoder.setDropLast(False)
ohe = marital_encoder.fit(df1)
df = ohe.transform(df1)
```

Line 1. Here we are initializing the OneHotEncoder objectmarital_encoder. The output column outCol marital_vector is going ot be used by the OneHotEncoder to store the vector that the OneHotEncoder object will create.
Line 2. We are setting the dropLast parameter of the OneHotEncoder to false so that we ca create a binary vector for each category including hte last one.
Line 3. ohe is now representing the the fitted encodded from the marital_encoder of df1 and now is ready for the data transformation based on the unique values found in the "marital_num" column of df1.

```
housing_indexer = StringIndexer(inputCol="housing", outputCol="housing_num").fit(df)
```

Here we are repeating the proccess of transforming the data to be suitable for our ML model

I created a StingIndexer called housing indexer to transform housing to housing_num.

```
df1 = housing_indexer.transform(df)
```

I applied the housing_indexer to df.

```
df1.show()
```

```
| single|     no|    1|       0.0| (2,[0],[1.0])|       0.0|
| single|    yes|    0|       0.0| (2,[0],[1.0])|       1.0|
| single|     no|    0|       0.0| (2,[0],[1.0])|       0.0|
| single|    yes|    0|       0.0| (2,[0],[1.0])|       1.0|
| single|     no|    1|       0.0| (2,[0],[1.0])|       0.0|
| single|    yes|    1|       0.0| (2,[0],[1.0])|       1.0|
| single|     no|    0|       0.0| (2,[0],[1.0])|       0.0|
|married|    yes|    0|       1.0| (2,[1],[1.0])|       1.0|
|married|     no|    1|       1.0| (2,[1],[1.0])|       0.0|
|married|    yes|    0|       1.0| (2,[1],[1.0])|       1.0|
|married|    yes|    0|       1.0| (2,[1],[1.0])|       1.0|
|married|    yes|    0|       1.0| (2,[1],[1.0])|       1.0|
|married|    yes|    0|       1.0| (2,[1],[1.0])|       1.0|
|married|    yes|    0|       1.0| (2,[1],[1.0])|       1.0|
|married|    yes|    0|       1.0| (2,[1],[1.0])|       1.0|
|married|    yes|    0|       1.0| (2,[1],[1.0])|       1.0|
|married|    yes|    0|       1.0| (2,[1],[1.0])|       1.0|
|married|    yes|    0|       1.0| (2,[1],[1.0])|       1.0|
+-------+-------+-----+----------+--------------+----------+
only showing top 20 rows
```

I am showing the dataframe df. df has three additional columns (marital_num, marital_vector, and housing_num) compared to df1.

```
housing_encoder = OneHotEncoder(inputCol="housing_num", outputCol="housing_vector")
housing_encoder.setDropLast(False)
ohe = housing_encoder.fit(df1) # indexer is the existing dataframe, see the question
df = ohe.transform(df1)
```

I created a OneHotEncoder named housing_encoder. I appied housing_encoder to df.

```
df1.show()
```

```
| single|     no|    1|        0.0| (2,[0],[1.0])|        0.0|
| single|    yes|    0|        0.0| (2,[0],[1.0])|        1.0|
| single|     no|    0|        0.0| (2,[0],[1.0])|        0.0|
| single|    yes|    0|        0.0| (2,[0],[1.0])|        1.0|
| single|     no|    1|        0.0| (2,[0],[1.0])|        0.0|
| single|    yes|    1|        0.0| (2,[0],[1.0])|        1.0|
| single|     no|    0|        0.0| (2,[0],[1.0])|        0.0|
|married|    yes|    0|        1.0| (2,[1],[1.0])|        1.0|
|married|     no|    1|        1.0| (2,[1],[1.0])|        0.0|
|married|    yes|    0|        1.0| (2,[1],[1.0])|        1.0|
|married|    yes|    0|        1.0| (2,[1],[1.0])|        1.0|
|married|    yes|    0|        1.0| (2,[1],[1.0])|        1.0|
|married|    yes|    0|        1.0| (2,[1],[1.0])|        1.0|
|married|    yes|    0|        1.0| (2,[1],[1.0])|        1.0|
|married|    yes|    0|        1.0| (2,[1],[1.0])|        1.0|
|married|    yes|    0|        1.0| (2,[1],[1.0])|        1.0|
|married|    yes|    0|        1.0| (2,[1],[1.0])|        1.0|
+-------+-------+-----+-----------+--------------+-----------+
only showing top 20 rows
```

```
df.show()
```

```
| single|     no|    1|        0.0| (2,[0],[1.0])|        0.0| (2,[0],[1.0])|
| single|    yes|    0|        0.0| (2,[0],[1.0])|        1.0| (2,[1],[1.0])|
| single|     no|    0|        0.0| (2,[0],[1.0])|        0.0| (2,[0],[1.0])|
| single|    yes|    0|        0.0| (2,[0],[1.0])|        1.0| (2,[1],[1.0])|
| single|     no|    1|        0.0| (2,[0],[1.0])|        0.0| (2,[0],[1.0])|
| single|    yes|    1|        0.0| (2,[0],[1.0])|        1.0| (2,[1],[1.0])|
| single|     no|    0|        0.0| (2,[0],[1.0])|        0.0| (2,[0],[1.0])|
|married|    yes|    0|        1.0| (2,[1],[1.0])|        1.0| (2,[1],[1.0])|
|married|     no|    1|        1.0| (2,[1],[1.0])|        0.0| (2,[0],[1.0])|
```

```
|married|    yes|    0|        1.0| (2,[1],[1.0])|        1.0| (2,[1],[1.0])|
|married|    yes|    0|        1.0| (2,[1],[1.0])|        1.0| (2,[1],[1.0])|
|married|    yes|    0|        1.0| (2,[1],[1.0])|        1.0| (2,[1],[1.0])|
|married|    yes|    0|        1.0| (2,[1],[1.0])|        1.0| (2,[1],[1.0])|
+-------+-------+-----+-----------+-------------+-----------+-------------+
only showing top 20 rows
```

Notice that here we have all of the columns that we have created so far for out transformation. marital_num, marital_vector, housing_num, housing vector.
As I understand it, the string indexer is transforming our non numerical columns to a column with numerical indices named _num then OneHotEncoder will create a binary vector in the columns _vector.

A new column housing_vector has been added to df.

```
# In Python
from pyspark.ml.feature import VectorAssembler
vecAssembler = VectorAssembler(inputCols=["marital_vector","housing_vector"], outputCol="features")
#VectorAssembler takes a list of input columns and creates a new DataFrame with an additional column
df = vecAssembler.transform(df)
df.select("marital_vector", "features", "housing_vector").show(10)
```

```
IllegalArgumentException: Output column features already exists.
```

I runned the code avobe by mistake and got that error message that I do understand. I just wanted save my cluster from diyng.... so I did run a df.show() better.

Line 2. We are importing the vectorAssembler class from the machine learning library pyspark.ml.feature
Line 3. Here are setting the instance VectorAssembler called vecAssembler with the _vector colummns as inputs and the features as the output. The features column will have the combined vectors for further statisticall analysis.
Line 5. Here we are applying the VectorAssembler to our df dataframe and the result df have the new feature columns
Line 6. we are selecting 3 columns "marital_vector", "features", and "housing_vector" and showing the results .show(10)

I created a new column called features using the input columns. I have selected the first 10 rows fo marital_vector, features, and housing_vectors.

```
df.select("marital_vector", "features", "housing_vector").show(10)
```

```
+--------------+-----------------+--------------+
|marital_vector|         features|housing_vector|
+--------------+-----------------+--------------+
| (2,[1],[1.0])|[0.0,1.0,1.0,0.0]| (2,[0],[1.0])|
| (2,[0],[1.0])|[1.0,0.0,0.0,1.0]| (2,[1],[1.0])|
| (2,[0],[1.0])|[1.0,0.0,1.0,0.0]| (2,[0],[1.0])|
| (2,[0],[1.0])|[1.0,0.0,0.0,1.0]| (2,[1],[1.0])|
| (2,[0],[1.0])|[1.0,0.0,1.0,0.0]| (2,[0],[1.0])|
| (2,[0],[1.0])|[1.0,0.0,0.0,1.0]| (2,[1],[1.0])|
| (2,[0],[1.0])|[1.0,0.0,1.0,0.0]| (2,[0],[1.0])|
| (2,[0],[1.0])|[1.0,0.0,0.0,1.0]| (2,[1],[1.0])|
| (2,[0],[1.0])|[1.0,0.0,1.0,0.0]| (2,[0],[1.0])|
| (2,[1],[1.0])|[0.0,1.0,0.0,1.0]| (2,[1],[1.0])|
+--------------+-----------------+--------------+
```

```
only showing top 10 rows
```

```
#df = df_assembler.transform(df)
```

```
df.show()
```

```
| single|     no|    1|        0.0| (2,[0],[1.0])|        0.0| (2,[0],[1.0])|[1.0,0.0,1.0,0.0]|
| single|    yes|    0|        0.0| (2,[0],[1.0])|        1.0| (2,[1],[1.0])|[1.0,0.0,0.0,1.0]|
| single|     no|    0|        0.0| (2,[0],[1.0])|        0.0| (2,[0],[1.0])|[1.0,0.0,1.0,0.0]|
| single|    yes|    0|        0.0| (2,[0],[1.0])|        1.0| (2,[1],[1.0])|[1.0,0.0,0.0,1.0]|
| single|     no|    1|        0.0| (2,[0],[1.0])|        0.0| (2,[0],[1.0])|[1.0,0.0,1.0,0.0]|
| single|    yes|    1|        0.0| (2,[0],[1.0])|        1.0| (2,[1],[1.0])|[1.0,0.0,0.0,1.0]|
| single|     no|    0|        0.0| (2,[0],[1.0])|        0.0| (2,[0],[1.0])|[1.0,0.0,1.0,0.0]|
|married|    yes|    0|        1.0| (2,[1],[1.0])|        1.0| (2,[1],[1.0])|[0.0,1.0,0.0,1.0]|
|married|     no|    1|        1.0| (2,[1],[1.0])|        0.0| (2,[0],[1.0])|[0.0,1.0,1.0,0.0]|
|married|    yes|    0|        1.0| (2,[1],[1.0])|        1.0| (2,[1],[1.0])|[0.0,1.0,0.0,1.0]|
|married|    yes|    0|        1.0| (2,[1],[1.0])|        1.0| (2,[1],[1.0])|[0.0,1.0,0.0,1.0]|
|married|    yes|    0|        1.0| (2,[1],[1.0])|        1.0| (2,[1],[1.0])|[0.0,1.0,0.0,1.0]|
|married|    yes|    0|        1.0| (2,[1],[1.0])|        1.0| (2,[1],[1.0])|[0.0,1.0,0.0,1.0]|
|married|    yes|    0|        1.0| (2,[1],[1.0])|        1.0| (2,[1],[1.0])|[0.0,1.0,0.0,1.0]|
|married|    yes|    0|        1.0| (2,[1],[1.0])|        1.0| (2,[1],[1.0])|[0.0,1.0,0.0,1.0]|
|married|    yes|    0|        1.0| (2,[1],[1.0])|        1.0| (2,[1],[1.0])|[0.0,1.0,0.0,1.0]|
|married|    yes|    0|        1.0| (2,[1],[1.0])|        1.0| (2,[1],[1.0])|[0.0,1.0,0.0,1.0]|
|married|    yes|    0|        1.0| (2,[1],[1.0])|        1.0| (2,[1],[1.0])|[0.0,1.0,0.0,1.0]|
+-------+-------+-----+-----------+-------------+-----------+-------------+-----------------+
only showing top 20 rows
```

I added a new column called features to df.

```
from pyspark.ml.stat import ChiSquareTest
```

I have imported ChiSquareTest from pyspark.

```
chi_sq = ChiSquareTest.test(df, "features", "label").head()
```

I am applying the Chi Square test on features and label.

```
print("pValues: " + str(chi_sq.pValues))
```

```
pValues: [0.5774606868568362,0.5774606868568362,0.02859622174508436,0.02859622174508436]
```

The P-values obtained from the Chi Square test are 0.5774606868568362, 0.5774606868568362, 0.02859622174508436, 0.02859622174508436.

What Databricks libraries did we use for this problem?

- pyspark.ml.feature
- pyspark.ml.stat

When do we use the Chi-square test method?

- We use chi-square to determine whether there's a statistically significant association between categorical variables. In the context of machine learning, it's typically used to test the independence of features and a categorical label.

What are the features in this problem?

- Marital
- Housing

What were the transformations applied?

- Transformed "marital" and " housing to numerical values
- Tranformed numerical values to binary vectors
- Applied VectorAssembler to binary vectors into a new DataFrame with column "features" added

What is the target variable in this problem?

- Label

What is the result of the Chi-square test? Interpret it.

- We identified that there is no correlation between marital status and defaulting on a loan (p-value 0.5774606868568362) because the p-value is bigger than 0.05 and the null hypotesis can be rejected.
  However, there is a correlation (p-value: 0.02859622174508436) between housing and defaulting on a loan because the p-value here is low and less than or equals to 0.05 therefore the null hypotesis cannot be rejected.
  It makes sense that there is a correlation with in the housing the defaulting variables because it takes a house to default on a house. But marriage has nothing to do with defaulting on a house.

I will now terminate the cluster