Table of Contents

[Other Regression Statistics in the Model Output 1](#_Toc30605940)

[Conditional Number (Collinearity) 2](#_Toc30605941)

[Omnibus Test (Normality Goodness of Fit) 2](#_Toc30605942)

[Handling Missing Data 2](#_Toc30605943)

[Techniques for Imputing During Linear Regression 3](#_Toc30605944)

[Imputing by Mean, Median or Mode 3](#_Toc30605945)

[Regression imputation 3](#_Toc30605946)

[Hot deck imputation 3](#_Toc30605947)

[Cold deck imputation 3](#_Toc30605948)

[Multiple imputation 3](#_Toc30605949)

[Imputation Case Study 3](#_Toc30605950)

[Dummy Variables for Categorical and Ordinal Variables 6](#_Toc30605951)

[Creating Binary Dummy Variables 7](#_Toc30605952)

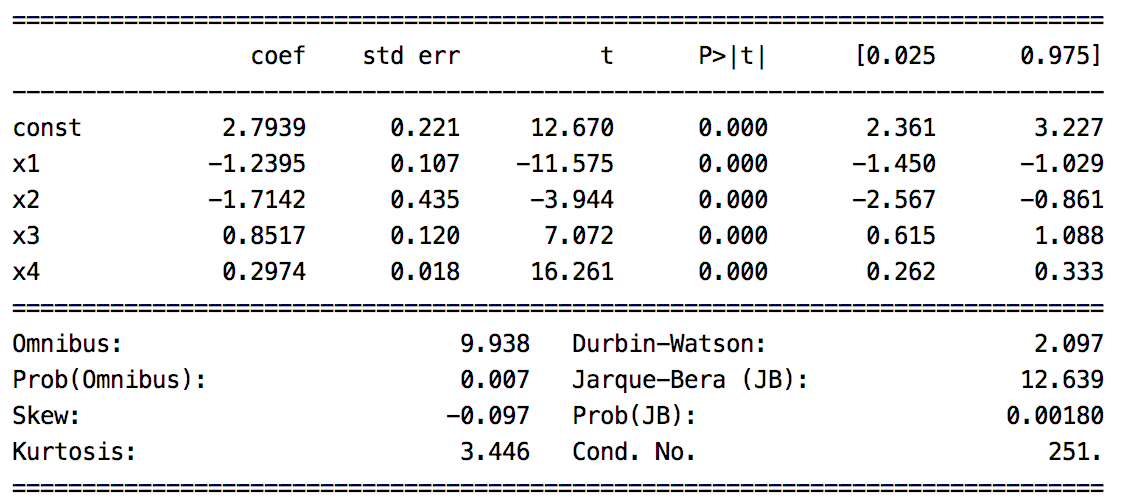
[get\_dummies() 8](#_Toc30605953)

[Bin Categories 9](#_Toc30605954)

## Other Regression Statistics in the Model Output

When developing OLS models, you will have noticed some other statistics at the end of the OLS regression model output (refer to Table 1).

Table 1: Additional Model Output Statistics



I will provide a brief summary of these statistics in this section.

#### Durbin Watson Test (Error residual independence)

The Durbin Watson test indicates whether or not the error residuals are independent of each other. Ideally the model has accounted for as much explainable variance as possible so the residual errors are independent of each other. Table 2 explains how to interpret the different Durbin-Watson ranges. A Durbin-Watson value around 2 is desirable.

Table 2: Ranges for Durbin-Watson

|  |  |
| --- | --- |
| **Range** | **Interpretation** |
| 0 to 2 | A positive correlation exists among residuals. |
| Around 2  *(Around 2 is ideal)* | Error residuals are independent (which is preferred). |
| 2 to 4 | A negative autocorrelation |

#### Jarque-Bera (Model normality)

The Jarque–Bera test is a goodness-of-fit test of whether a model has the skewness and kurtosis matching a normal distribution. The null hypothesis, , suggests that the distribution is normal where the Kurtosis, or height, is approximately 3 and skewness is 0. When the p-value > 0.05 accept the null hypothesis. With samples below 2000, JB is prone to rejecting the null hypothesis. For the case presented in Table 1, normality is rejected.

### Conditional Number (Collinearity)

The conditional number measures likelihood for collinearity or unwanted interdependence between predictor variables. Ideally predictor variables are independent. High interdependence means higher variance in results from our model.

Linear regression models are prone to higher amounts of collinearity. If the models have high collinearity this should be documented and explained to a subject matter expert who is familiar with the model variables and the goals of the model. Sometimes high collinearity is acceptable. For example, SAT and LSAT scores on college entrance exams are highly correlated. Both indicate that the applicant is prepared for a rigorous academic program. Collinearity, could be high for these predictor variables yet we may still want both variables in the model.

Other times, high collinearity may be unacceptable. For example, a stock portfolio may rely on a complex relationship between sectors. A portfolio model may be incorrectly constructed under the assumption that oil and housing prices rise together yet an unexpected change in one of these sectors could have a detrimental effect on the prediction.

### Omnibus Test (Normality Goodness of Fit)

The Omnibus tests examines the model for goodness of fit with a normal distribution.

It is normally distributed.

Since the p-value is small in Table 1 we would reject the null hypothesis.

## Handling Missing Data

Often, when you prepare your data for modeling you will need to handle missing values because the algorithms require numbers. You can handle missing values in many ways and none of them are ideal.

* You can remove the columns or rows but this practice is almost never recommended. Sometimes the data is missing for a reason. Removing items can produce a bias in the model. A case where removing data may make sense could be when only 10 or 20% of the attributes are present for a particular column of a data set.
* You can impute data for missing values. Imputing may not necessarily improve the model either and this practice can introduce bias to your model.

In spite of this challenge, keeping the data and imputing missing values is most common.

### Techniques for Imputing During Linear Regression

This section suggests ways that you may impute content. You may create other methods to impute values as well or you may build a method for imputing that uses a hybrid of techniques.

For now, you may keep your imputing methods simple but they could become very advanced if needed.

Often imputing techniques do not offer much lift in the regression results. On the other hand, I have occasionally experienced significant performance gains through imputing. In any case, the topic is a necessary one and it can make a positive difference when done properly.

### Imputing by Mean, Median or Mode

For linear regression algorithms you can impute missing values by providing the sample mean, median or mode. This technique can be implemented quickly but it also reduces the model variance. Be wary that bias and inaccuracy may result.

### Regression imputation

Missing values can be predicted with simple OLS models based on other variables. This preserves relationships among variables in the imputation model. This technique can become complex if predictor variables are also missing data.

### Hot deck imputation

Hot deck imputation involves selecting the missing value from a sample has that similar values for other variables. Or, compute the average value from a random selection of similar samples.

### Cold deck imputation

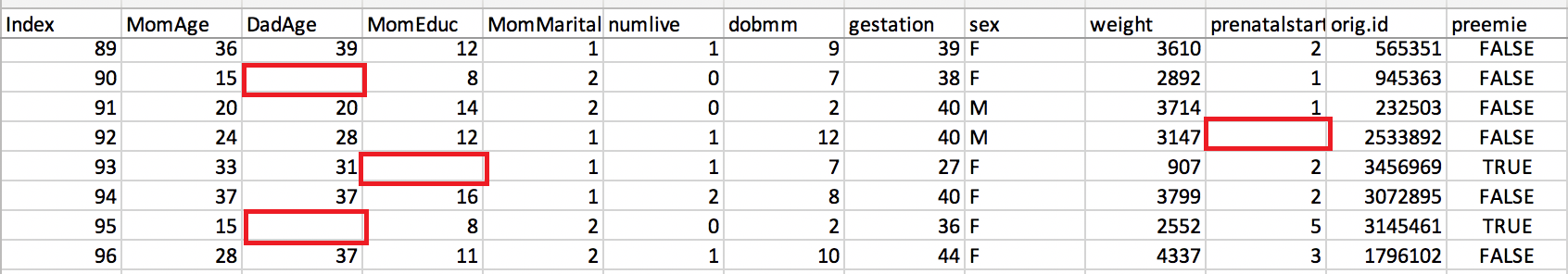
Cold deck imputation is a systematically chosen value from a sample that has similar values on other variables. Cold deck imputation is similar to hot deck imputation but it removes the random variation. For example, you may always choose the third individual in the same experimental condition and block.

### Multiple imputation

You can use more than one method to impute values. Because these methods have a random component, the multiple estimates are slightly different. This re-introduces some variation that your software can incorporate in order to give your model accurate estimates of standard error. Multiple imputation solves a lot of problems with missing data (though, unfortunately not all) and if done well, leads to unbiased parameter estimates and accurate standard errors. You may choose to average or weight the results to generate a single result based on multiple methods.

### Imputation Case Study

When inspecting the babysamp-98.txt file we can see several missing values exist.

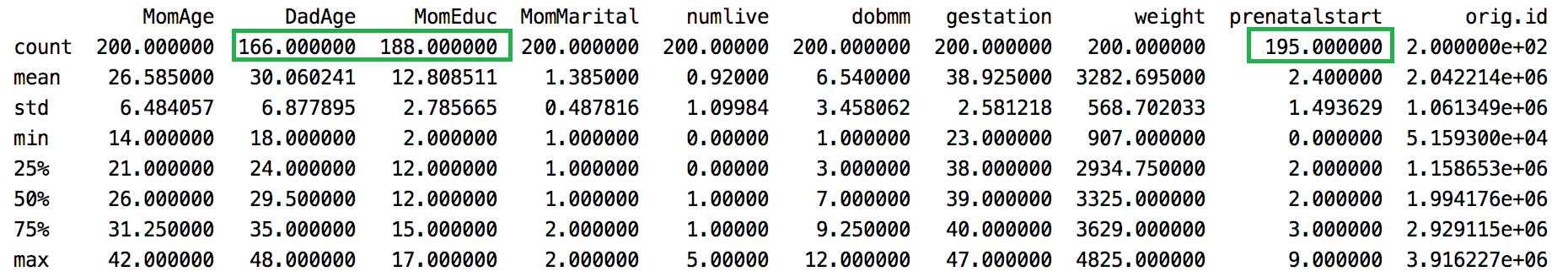


Sometimes opening the file and finding all columns with missing values is not so simple so you can use the *describe()* command to generate a numerical summary of non-null counts.

print(df.describe()) *# View counts that highlight missing values.*

When inspecting descriptive statistics for numeric data in the set we can see that *DadAge*, *MomEduc* and *prenatalstart* have missing values (see Table 3) since their row counts are lower than 200.

Table 3: Using describe() to find missing values

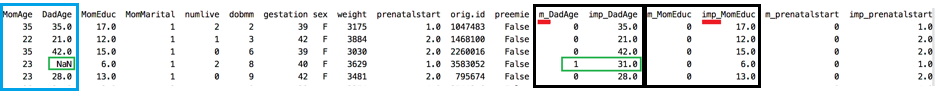


Example 1: Imputing with Mean, Median and Mode

Sometimes values are missing for a reason. In this case the Father’s age is missing and the Mother’s education level is missing. These values could be missing perhaps because the Father is absent or the Mother is not willing to report her education level. Or the person collecting the data may have recorded the data differently than others who collect or interpret data. We can speculate but the fact that data is missing could be relevant to our model.

This example demonstrates how to not only create columns with imputed values, another column is created to indicate if the value was imputed. In the end, for every missing value two other columns are created. Notice how **m\_** column indicates if the value in the **imp\_** column is imputed or not (refer to Table 4). Also note that the original column is retained.

Table 4: Imputed Values for DadAge



In the end, you can use either **m\_** or **imp\_** columns in your regression – or both. For this case the new predictors do not offer useful performance gain but the technique can sometimes be extremely useful. Here is the code to impute the values and perform the regression.

|  |
| --- |
| import pandas as pd  import numpy as np  from sklearn.model\_selection import train\_test\_split  import statsmodels.api as sm  from sklearn import metrics  # Import data into a DataFrame.  path = "/Users/pm/Desktop/DayDocs/2019\_2020/PythonForDataAnalytics/workingData/babysamp-98.txt"  df = pd.read\_table(path, skiprows=1,  delim\_whitespace=True,  names=('MomAge', 'DadAge', 'MomEduc', 'MomMarital', 'numlive',  "dobmm", 'gestation', 'sex', 'weight', 'prenatalstart',  'orig.id', 'preemie'))  # Show all columns.  pd.set\_option('display.max\_columns', None)  pd.set\_option('display.width', 1000)  print(df.head()) # View a snapshot of the data.  print(df.describe()) # View stats including counts which highlight missing values.  def convertNAcellsToNum(colName, df, measureType):  # Create two new column names based on original column name.  indicatorColName = 'm\_' + colName # Tracks whether imputed.  imputedColName = 'imp\_' + colName # Stores original & imputed data.  # Get mean or median depending on preference.  imputedValue = 0  if(measureType=="median"):  imputedValue = df[colName].median()  elif(measureType=="mode"):  imputedValue = float(df[colName].mode())  else:  imputedValue = df[colName].mean()  # Populate new columns with data.  imputedColumn = []  indictorColumn = []  for i in range(len(df)):  isImputed = False  # mi\_OriginalName column stores imputed & original data.  if(np.isnan(df.loc[i][colName])):  isImputed = True  imputedColumn.append(imputedValue)  else:  imputedColumn.append(df.loc[i][colName])  # mi\_OriginalName column tracks if is imputed (1) or not (0).  if(isImputed):  indictorColumn.append(1)  else:  indictorColumn.append(0)  # Append new columns to dataframe but always keep original column.  df[indicatorColName] = indictorColumn  df[imputedColName] = imputedColumn  return df  df = convertNAcellsToNum('DadAge', df, "mode")  df = convertNAcellsToNum('MomEduc', df, "mean")  df = convertNAcellsToNum('prenatalstart', df, "median")  print(df.head(10))  # You can include both the indicator 'm' and imputed 'imp' columns in your model.  # Sometimes both columns boost regression performance and sometimes they do not.  X = df[[ 'gestation', 'm\_DadAge', 'imp\_DadAge', 'm\_prenatalstart', 'imp\_prenatalstart',  'm\_MomEduc', 'imp\_MomEduc']].values  # Adding an intercept \*\*\* This is requried \*\*\*. Don't forget this step.  X = sm.add\_constant(X)  y = df['weight'].values  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=0)  # Build and evaluate model.  model = sm.OLS(y\_train, X\_train).fit()  predictions = model.predict(X\_test) # make the predictions by the model  print(model.summary())  print('Root Mean Squared Error:', np.sqrt(metrics.mean\_squared\_error(y\_test, predictions))) |

The output shows the m\_ and imp\_ columns for DadAge as shown in Table 4.

## Dummy Variables for Categorical and Ordinal Variables

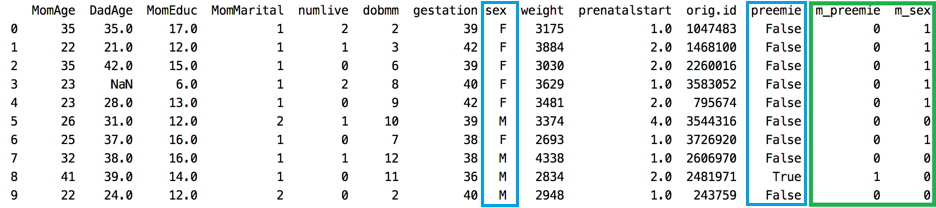
Regression analysis treats all independent (X) variables in the analysis as numerical. Numerical variables are interval or ratio scale variables whose values are directly comparable, e.g. ‘10 is twice as much as 5’, or ‘3 minus 1 equals 2’. Often, however, you might want to include an attribute or nominal scale variable such as ‘Product Brand’ or ‘Type of Defect’ in your study. Say you have three types of defects, numbered ‘1’, ‘2’ and ‘3’. In this case, ‘3 minus 1’ doesn’t mean anything. The numbers here are used to indicate or identify the levels of ‘Defect Type’ and do not have intrinsic meaning of their own. Dummy variables are created in this situation to ‘trick’ the regression algorithm into correctly analyzing attribute variables. The dummy variables act like ‘switches’ that turn various parameters on and off in an equation.

### Creating Binary Dummy Variables

Example 2: Creating Binary Dummy Variables

For this example, we are creating binary dummy columns which have either 0 or 1 as values. Notice that the columns begin with **m\_** + the original name. The *sex* column contains either M or F but we need it to be 0 or 1 respectively. The *preemie* column contains False or True but again our regression algorithm requires 0 or 1 respectively (see Table 5).

Table 5: Imputed Binary Values Are Usually Always Either 0 or 1



Here is the code used to generate the output in Table 5:

|  |
| --- |
| import pandas as pd  import numpy as np  from sklearn.model\_selection import train\_test\_split  import statsmodels.api as sm  from sklearn import metrics  # Import data into a DataFrame.  path = "/Users/pm/Desktop/DayDocs/2019\_2020/PythonForDataAnalytics/workingData/babysamp-98.txt"  df = pd.read\_table(path, skiprows=1,  delim\_whitespace=True,  names=('MomAge', 'DadAge', 'MomEduc', 'MomMarital', 'numlive',  "dobmm", 'gestation', 'sex', 'weight', 'prenatalstart',  'orig.id', 'preemie'))  # Show all columns.  pd.set\_option('display.max\_columns', None)  pd.set\_option('display.width', 1000)  print(df.head()) # View a snapshot of the data.  print(df.describe()) # View stats including counts which highlight missing values.  def createBinaryDummyVar(df, colName, val0, val1):  imputedColName = "m\_" + colName  imputedCol = []  for i in range(len(df)):  if(df.loc[i][colName]==val0):  imputedCol.append(0)  else:  imputedCol.append(1)  df[imputedColName] = imputedCol  return df  df = createBinaryDummyVar(df, "preemie", False, True)  df = createBinaryDummyVar(df, "sex", "M", "F")  print(df.head(10)) |

The output generates the new DataFrame structure that is presented in Table 5.

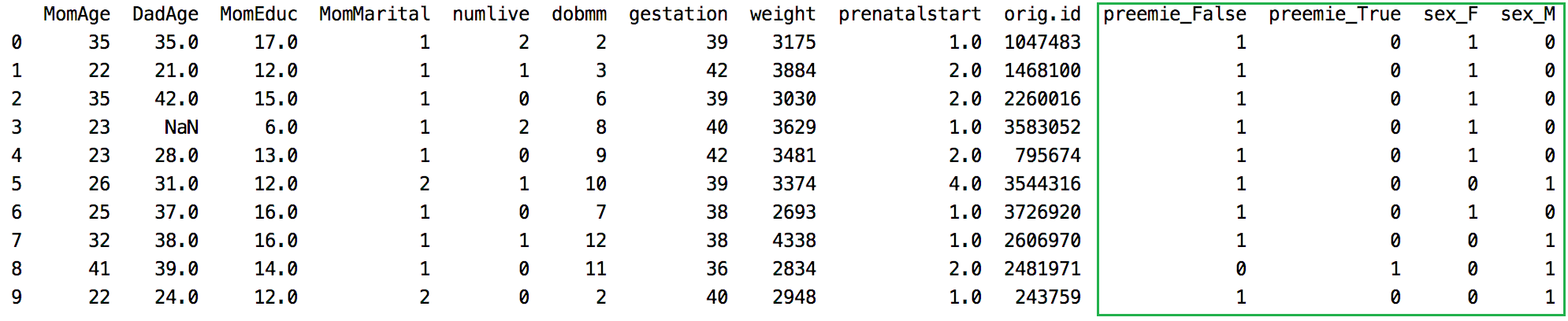
### get\_dummies()

The get\_dummies() method automates the generation of dummy variables.

Example 3: A Faster Way to Create Dummy Variables from Categories

This example demonstrates how to use ***get\_dummies()*** to quickly convert ‘sex’ and ‘preemie’ columns into numeric dummy variable columns (refer to Table 6). This code is much easier to implement than the code in Example 2.

Table 6: Auto-Generated Dummy Variable Columns



Here is the code:

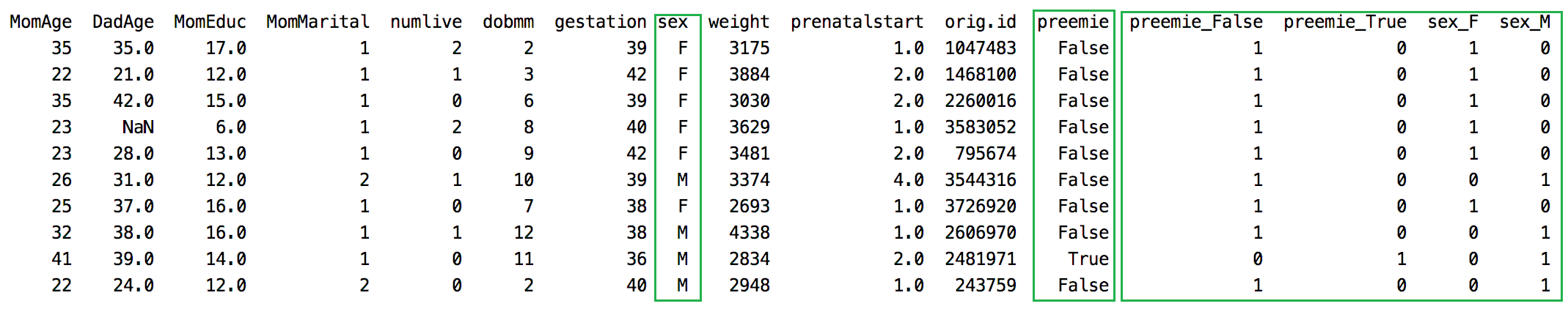
|  |
| --- |
| **import** pandas **as** pd **import** numpy **as** np **from** sklearn.model\_selection **import** train\_test\_split **import** statsmodels.api **as** sm **from** sklearn **import** metrics  *# Import data into a DataFrame.* path = **"/Users/pm/Desktop/DayDocs/2019\_2020/PythonForDataAnalytics/workingData/babysamp-98.txt"** df = pd.read\_table(path, skiprows=1,  delim\_whitespace=**True**,  names=(**'MomAge'**, **'DadAge'**, **'MomEduc'**, **'MomMarital'**, **'numlive'**,  **"dobmm"**, **'gestation'**, **'sex'**, **'weight'**, **'prenatalstart'**,  **'orig.id'**, **'preemie'**)) *# Show all columns.* pd.set\_option(**'display.max\_columns'**, **None**) pd.set\_option(**'display.width'**, 1000) print(df.head()) *# View a snapshot of the data.* print(df.describe()) *# View stats including counts which highlight missing values.* df = pd.get\_dummies(df, columns=[**'preemie'**, **'sex'**]) print(df.head(10)) |

The results show the DataFrame that is shown in Table 6.

Example 4: Auto-Generated Dummy Variable Columns with Retained Original Columns

On its own, the *get\_dummies()* method removes the original column from the DataFrame. An adjustment can be made to retain the original column to maintain explanatory ability in the DataFrame. This example shows how to adjust the code in Example 3 to also retain the original *sex* and *preemie* columns after adding dummy variable columns. Generally, you do not want to delete original columns from your DataFrame since you will often want to refer back to the original data structure. Using this technique is quick, simple and extremely informative.

Table 7: Using get\_dummies() and retaining the original Columns



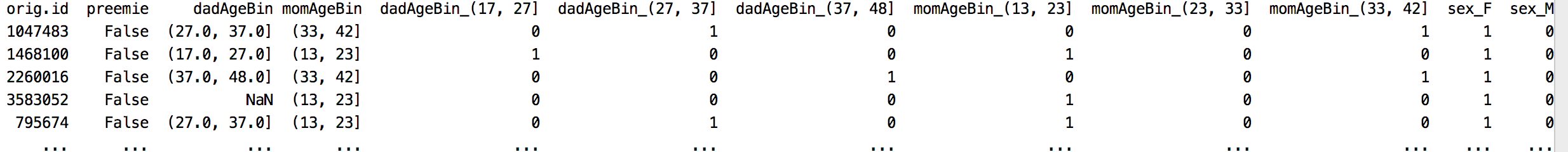
To build this replace the get\_dummies() instruction in Example 3 with these three lines.

|  |
| --- |
| tempDf = df[['preemie', 'sex']] # Isolate columns  dummyDf = pd.get\_dummies(tempDf, columns=['preemie', 'sex']) # Get dummies  df = pd.concat(([df, dummyDf]), axis=1) # Join dummy df with original df |

The output shows the original columns along with the dummy variable columns (see Table 7).

## Binning Categories

You may find that a variable like MomAge is insignificant on its own but if you bin it within clusters of 10 years the variables created may become significant. The DataFrame’s *cut()* function allows you to quickly split ordinal data into bins. Table 8 shows ages for Mom’s and Dad’s split into separate sub-groups. The sub-groups are labeled with bin notation. dadAgeBin\_(27,37] refers to ages 28 to 87 inclusively. The round bracket on the left does not include the number that it precedes. Table 8 shows separate bins that have been created for Mother and Father age groups.

Table 8: Bin Categories for MomAge and DadAge  


|  |
| --- |
| import pandas as pd  # Import data into a DataFrame.  path = "/Users/pm/Desktop/DayDocs/2019\_2020/PythonForDataAnalytics/workingData/babysamp-98.txt"  df = pd.read\_table(path, skiprows=1,  delim\_whitespace=True,  names=('MomAge', 'DadAge', 'MomEduc', 'MomMarital', 'numlive',  "dobmm", 'gestation', 'sex', 'weight', 'prenatalstart',  'orig.id', 'preemie'))  # Show all columns.  pd.set\_option('display.max\_columns', None)  pd.set\_option('display.width', 1000)  print(df.head()) # View a snapshot of the data.  print(df.describe()) # View stats including counts which highlight missing values.  #print("\nbinned\_statistic for median : \n", stats.binned\_statistic(df['DadAge'].values, bins = 4))  df['dadAgeBin'] = pd.cut(x=df['DadAge'], bins=[17, 27, 37, 48])  df['momAgeBin'] = pd.cut(x=df['MomAge'], bins=[13,23,33,42])  tempDf = df[['dadAgeBin','momAgeBin','sex']] # Isolate columns  # Get dummies  dummyDf = pd.get\_dummies(tempDf, columns=['dadAgeBin','momAgeBin','sex'])  df = pd.concat(([df, dummyDf]), axis=1) # Join dummy df with original  print(df) |

The output from running this example generates the content shown in Table 8.

Exercise (4 marks)

Read the homework 1 data into a DataFrame. Examine the data for the homework with the instructions:

* print(df.head().transpose())
* print(df.describe().transpose())

Impute values for ﻿OverallQual. Retain the original column and also create an m\_OverallQual variable to indicate if the data has been imputed or not. Ensure the data has been imputed by inspecting the count with:

print(df.describe().transpose())

Show your code here:

|  |
| --- |
|  |

Show a screenshot of your describe().transpose() output after imputing:

|  |
| --- |
|  |

Exercise (4 marks)

Create binned columns to group one of the numeric columns. Show your code here:

|  |
| --- |
|  |

Show a screenshot of your DataFrame output with the instruction print(df.head().transpose())

|  |
| --- |
|  |

Exercise (4 marks)

Create dummy variables for a non-numeric column in your homework 1 data set. Retain the original column. Show your code here:

|  |
| --- |
|  |

Show a screenshot of your new columns here:

|  |
| --- |
|  |