Machine Learning End to End Flow Summary

The blog assumes the reader knows some basic Machine Learning terms.

This Summary post should be read along with the below blog post

<https://mrg-ai.github.io/blog/2020/09/20/ML_EndToEnd_usingCAHousingDataset.html>

# Get the data

* This could be via files or from a database. Try to get the data into a pandas dataframe.
* That is the best format to do further data exploration.
* Use head(), info(), describe() methods to look at the data and column/column type information
* Using matplotlib visualize the data via histograms for numerical columns
* For Categorical columns, find the unique values using value\_counts()

# Split the data into Training and Test

* Splitting the data can be random using a random seed using sklearns test\_train\_split class
* Get the labels into a different dataframe for using during prediction.
* However, in reality we should split using some unique column in the data (if available)
* We can create such column if that is possible by using some combination of existing columns.
* Sometimes, we may also have to do a stratified data split i.e. data should be taken from the different “strata” of the data. For example – Low Income Grp, High Income Grp etc.
  + StratifiedShuffleSplit class can be used along with some column which indicates the “strata”

# Visualize and Data Cleanup of the training data

* Use matplotlib and look at the training data in more detail.
* Create new attributes (new dataframe columns) from existing columns if possible
* Date columns can definitely be used to create new Day, Month, Quarter, Year etc (DatePart function)
* Clean the data for NULLs, Blanks etc through any of the below methods. Each method has its own implications and should be considered appropriately.
  + Drop such rows
  + Use sklearn SimpleImputer to impute such NULLs with Median or Mean values
* **NOTE –** SimpleImputer works only on Numerical columns and CategoricalEncoders work on Categorical columns. We generally create different dataframes for Numericals and Categoricals. There is another Class which can handle both together and we will look at it in next section.
* Handle the Categorical values i.e. convert them to Numbers
  + Ordinal Encoder Class for Ordinal (they have a relationship between them) Categories like *Low, Medium, High*
  + One Hot Encoder Class for categories which are Nominal (unrelated to one another) like List of State Names (CA, AZ, NY etc.)
  + We can also create Custom Transformers to do some custom transformations. An example below. Using BaseEstimator and TransformerMixin classes we can get many existing sklearn methods in the Custom Transformer.

from sklearn.base import BaseEstimator**,** TransformerMixin

rooms\_ix**,** bedrooms\_ix**,** population\_ix**,** households\_ix **=** 3**,** 4**,** 5**,** 6

**class** **CombinedAttributesAdder(**BaseEstimator**,** TransformerMixin**):**

**def** \_\_init\_\_**(***self***,** add\_bedrooms\_per\_room**=True):** *# no \*args or \*\*kargs*

*self***.**add\_bedrooms\_per\_room **=** add\_bedrooms\_per\_room

**def** **fit(***self***,** X**,** y**=None):**

**return** *self* *# nothing else to do*

**def** **transform(***self***,** X**):**

rooms\_per\_household **=** X**[:,** rooms\_ix**]** **/** X**[:,** households\_ix**]**

population\_per\_household **=** X**[:,** population\_ix**]** **/** X**[:,** households\_ix**]**

**if** *self***.**add\_bedrooms\_per\_room**:**

bedrooms\_per\_room **=** X**[:,** bedrooms\_ix**]** **/** X**[:,** rooms\_ix**]**

**return** np**.**c\_**[**X**,** rooms\_per\_household**,** population\_per\_household**,**

bedrooms\_per\_room**]**

**else:**

**return** np**.**c\_**[**X**,** rooms\_per\_household**,** population\_per\_household**]**

attr\_adder **=** CombinedAttributesAdder**(**add\_bedrooms\_per\_room**=False)**

housing\_extra\_attribs **=** attr\_adder**.**transform**(**housing**.**values**)**

# Feature Scaling

* This is required because ML algorithms require the different numerical features to be on similar scales.
* Therefore, it is always a good practice to Scale the Numerical features.
* Min-Max Scaling (Normalization) can be achieved with MinMaxScaler Class. This will cause issues when the data has many outliers.
* Standardization can be achieved through StandardScaler Class. Less affected by outliers.

# Pipelines

* Since there are multiple steps we do as part of Data preprocessing, we should create a pipeline to do these transformations one after the another.
* The output of one becomes input to another and so on.
* A pipeline for numerical attributes can look like below

num\_pipeline **=** Pipeline**([**

**(**'imputer'**,** SimpleImputer**(**strategy**=**"median"**)),**

**(**'attribs\_adder'**,** CombinedAttributesAdder**()),**

**(**'std\_scaler'**,** StandardScaler**()),**

**])**

* We can get the different transformations done for numerical columns as below using the pipeline defined above
  + Dataframe\_Transformed = num\_pipeline.fit\_transform(Dataframe[numerical attribute list])
* We can create a Full Pipeline for all attributes at once also. This would be ideal instead of having separate pipelines for Numericals and Categoricals.
* Below is such an example. We can use ColumnTransformer Class. Remainder keyword is to tell that any columns not covered in the num pipeline or cat pipeline should be passed through.

from sklearn.compose import ColumnTransformer

num\_attribs **=** *list***(**dataframe\_num**)**

cat\_attribs **=** *list***(**dataframe\_cat**)**

num\_attribs

[‘List of Numeric Attributes’]

cat\_attribs

[‘List of Categorical Attributes’]

full\_pipeline **=** ColumnTransformer**([**

**(**"num"**,** num\_pipeline**,** num\_attribs**),**

**(**"cat"**,** OneHotEncoder**(),** cat\_attribs**),**

**],** remainder**=**'passthrough'**)**

* Dataframe\_AllCols\_Prepared = full\_pipeline.fit\_transform(Dataframe\_AllCols)
* The data is now ready to be trained.

# Evaluate Different Models

* Evaluate the different models
* Calculate the cross validations score using K Fold Cross Validation
* Pick the model which best suits the data.
* Save the picked model using joblib.dump(model, ‘<model\_name>.pkl') as a pkl file.
* This can be later reloaded back using joblib.load()

# Finetune the Model

* Finetune the models hyperparameters to finetune the model
* Using GridSearchCV or RandomizedSearchCV classes, we can get the best hyperparameters for the model.

# Deploy the Model

* The model can be deployed on Cloud or can be exposed through a REST API and the model’s predict function can be used to evaluate the output by giving the necessary inputs.
* Note that the test input or any new input has to be transformed using the same data preprocessing pipeline before the model can predict the output using that input.

X\_test\_prepared **=** full\_pipeline**.**transform**(**X\_test**)**

final\_predictions **=** final\_model**.**predict**(**X\_test\_prepared**)**