**End-to-End Machine Learning Project**

# Checklist for Machine Learning Projects

1. **Frame the problem and look at the big picture**
2. **Get the data**
3. **Explore the data to gain insights**
4. **Prepare the data for Machine Learning algorithms**
5. **Explore the Training Set data to gain insights**
6. **Prepare the data for Machine Learning algorithms**
7. **Explore many different models and short-list the best ones**
8. **Fine-tune model**
9. **Evaluate Model on Test Set**
10. **Present the solution**
11. **Launch, monitor, and maintain the system**
12. Frame the problem and look at the big picture

* What is Business objective & *expecting the* benefit from the Model
* Which Algorithm needs to be selected. : Supervised, Unsupervised or Reinforcement/Classification or Regression/Batch or Online Learning
* Performance Measure : MSE,RMSE etc.
* Efforts required to tweaking it.

1. Get the data

*Type 1:*

df = pd.read\_csv(‘housing.csv’)

*Type 2:*

Import os

def file\_read(f\_path,f\_name):

file\_path = os.path.join(f\_path,f\_name)

return pd.read\_csv(file\_path)

File\_loc = ‘C:/Users/Admin/Documents/machine\_learning/datasets/housing/’

File\_name = ‘housing.csv’

df = file\_read(File\_loc,File\_name)

1. Explore the data to gain insights

* df.head() : Check all the values & try to find the different types of values(like Categorical Values) or measurement of values(like 10000 scaled to 10)
* df.describe()/df.info() : Check the Count/types/mean/std/max etc. all of columns.
* Plot the Histogram to get the feel of Data.

df.hist(bins=50,figsize=(20,15))

df['median\_income'].hist()

**Check for Capped values or Long tail or Scaled Values.**

**🡪 If Values are Capped, then we need to think for estimate of those Capped Areas.**

**Solution :** Recollect the Data for those Capped Label or remove it from test Dataset.

**🡪 If Long tails are there, transform attribute to more bell-shaped Distributions.**

df['income\_cat'] = np.ceil(df['median\_income']/1.5)

df[['income\_cat'].where(df['income\_cat']<5,5.0,inplace=True)

**🡪 If Categorical Data are there, we can check the Count of each type**

val\_count = df['Categorical'].value\_counts()

1. Prepare the data for Machine Learning algorithms

* **Split the Data into Train & Test Set :** Split the Train & Test Set earliest for avoiding Data Snooping or Overfitting & again visualize the Training Set Data Again.

*Type 1: (Use Standard Method to split)*

from sklearn.model\_selection import train\_test\_split

train\_set, test\_set = train\_test\_split(df,test\_size=0.2,random\_state=42)

print(“Length of Training Set : ”,len(train\_set),” length of Test Set : ”,len(test\_set))

*Type 2: (User Defined Method to split)*

def split\_train\_test(data,test\_size):

np.random.seed(42)

shuffled\_indices = np.random.permutation(len(data))

test\_indices = int(len(data)\*test\_size)

test\_set = df[:test\_indices]

train\_set=df[test\_indices:]

return df.iloc[train\_set],df.iloc[test\_set]

**Problem with this Split Function :** When new records got added to the Dataset, splitting of data can cause Data Snooping.

*Type 3: (User Defined Method Using Identifer to split)*

import hashlib

def test\_set\_check(identifier,test\_size,hash):

return hash(np.int64(identifier)).digest()[-1] < 256\*test\_size

def split\_train\_test\_by\_id(data,identifier,test\_size,hash=hashlib.md5):

ids = data[identifer]

in\_test\_set = ids.apply(lambda id\_:test\_set\_check(id\_,test\_size,hash))

return data.loc[~in\_test\_set],data.loc[in\_test\_set]

**#Creating Unique ID & Calling the Method**

df\_id = df.reset\_index()

df\_id[‘id’] = df[‘longitude’]\*1000+df[‘latitude’]

train\_set,test\_set = split\_train\_test\_by\_id(df,id,0.2)

**Problem with this random Split Function :** Sampling Bias.

**Three Sampling Method :** ***Simple Random Sampling, Stratified Sampling, Cluster Sampling.*** In Stratified sampling , equal ratio of data is selected from each Cluster whereas in cluster Sampling, randomly select few cluster & then Randomly select the data.

**If any of the Column is selected for clustering in Stratified Sampling & that column is having many instances, then we need to limit the instances/Strata.**

df[‘income\_cat’] = np.ceil(df[‘median\_income’]/1.5)

df[‘income\_cat’].where(df[‘income\_cat’]<5,5.0,inplace=True)

df[‘income\_cat’].value\_counts()

*Type 4: (Startified Sampling to split)*

from sklearn.model\_selection import StratifiedShuffleSplit

split = StratifiedShuffleSplit(n\_splits=1,test\_size=0.2,random\_state=42)

for train\_index,test\_index in split.split(df,df[‘income\_cat’]):

strat\_train\_set = df.loc[train\_index]

strat\_test\_set = df.loc[test\_index]

**# Compare income category proportion in Stratified Sampling and Random Sampling**

**#Calculate Proportion**

def income\_prop\_cal(data):

return data[‘income\_cat’].value\_counts()/len(data)

**# Comparison**

Compare\_prop = pd.Dataframe({

“Overall ” : income\_prop\_cal (housing),

“Random” : income\_prop\_cal (test\_set),

“Strata” : income\_prop\_cal(strat\_test\_set),

}).sort\_index()

Compare\_prop[“Random % Error”] = 100 \* Compare\_prop[‘Random’]/ Compare\_prop[‘Overall’]-100

Compare\_prop[“Strata % Error”] = 100 \* Compare\_prop[‘Strata]/ Compare\_prop[‘Overall’]-100

**Stratified Sampling gives better test set than Random Sampling**

**🡪 Once Split happened, better to delete the newly added column (used for Stratified Sampling) to get the Original State.**

**#Type 1 :**

Strat\_train\_set.drop(‘income\_cat’,axis=1,inplace=True)

Strat\_test\_set.drop(‘income\_cat’,axis=1,inplace=true)

**#Type 2:**

for set\_ in (strat\_train\_set,strat\_test\_set):

set\_.drop(‘income\_cat’,axis=1,inplace=True)

1. Explore the Training Set data to gain insights

* **Create a copy of Training Set Data**

housing = strat\_train\_set.copy()

🡪 **Visualize the Data**

#Get to know the High/Low Density

housing.plot(kind=’scatter’,x=’longitude’,y=’latitude’,alpha=0.1)

**#More Depth**

housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.4,

s=housing["population"]/100, label="population", figsize=(10,7),

c="median\_house\_value", cmap=plt.get\_cmap("jet"), colorbar=True,

sharex=False)

plt.legend()

**🡪 Check the Correlations**

Cor = housing.corr()

Cor[‘median\_house\_value’].sort\_values(ascending = False)

**#We can find the Features which are Positive/Negative Corelated with the TARGET Column.**

**🡪 Check the Most Correlations Columns Using SCATTER\_MATRIX**

from pandas.plotting import scatter\_matrix

attributes = ["median\_house\_value", "median\_income", "total\_rooms","housing\_median\_age"]

scatter\_matrix(housing[attributes],figsize=(12,8))

🡪 **Visualize most promising Attribute**

housing.plot(kind=’scatter’,x=’median\_income’,y=’housing\_median\_age’)

plt.axis([0,16,0,55000])

🡪 **Sometimes some columns may not look useful, we can experiment with those columns tries to find whether any useful columns can be created.**

**#Experiment with other Attributes Combinations**

housing['rooms\_per\_houshold'] = housing['total\_rooms']/housing['households']

housing['bedrooms\_per\_room'] = housing['bedrooms']/housing['total\_rooms']

housing['population\_per\_household'] = housing['population']/housing['households']

* **Again, find the Correlation**

new\_cor = housing.corr()

new\_cor[‘median\_housing\_price’].sort\_values(ascending=False)

If we find any useful Attribute, we can add it to our training set later

1. Prepare the data for Machine Learning algorithms

* **Revert the Data to a clean Training Set**

housing = strat\_train\_set.drop('median\_house\_value',axis=1)

housing\_labels = strat\_train\_set['median\_house\_value'].copy()

Drop creates a copy of the Set, so it will not be affected by the changes.

* **Data Cleaning**
  + **Handling Missing Values**

Sample = housing[housing.isnull().any(axis=1)]

* + - ***Solution :***
      1. ***Drop the Missing Rows***

Sample.dropna(subset[‘total\_bedroom’],inplace=True)

* + - 1. **Drop the Attributes/Columns**

sample.drop(‘total\_bedroom’,axis=1)

* + - 1. **Fill the Missing Values**

median = housing[‘total\_bedroom’].median()

sample[‘total\_bedroom’].fillna(value=median,inplace=True)

* + - ***Missing Values Using Scikit-Learn Imputer Class***

🡪Median Cannot be calculated upon Categorical Values, so we need to create a copy of training set without Categorical Columns

🡪Since in the LIVE Model, missing values can come in any of the Columns, so we need to pass all the columns.

housing\_num = housing.drop(‘Ocean\_proximity’,axis=1)

from sklearn.preprocessing import Imputer

imputer = Imputer(missing\_values=’NaN’,strategy=’median’)

imputer.fit(housing\_num)

X = imputer.transform(housing\_num)

imputer.statistics\_ **#Median for all the columns**

* + **Handling Categorical Values**
    - 1. ***Type 1: factorize()***

# Convert ocean\_proximity to numbers

housing\_cat = housing[‘Ocean\_proximity’]

housing\_cat\_encoded, housing\_categories = housing\_cat.factorize()

housing\_cat\_encoded

housing\_categories

🡪**Problem** : Machine Learning algorithm may assume that two nearby values are more similar than two distant values as it will 0,1,2 values. So better ways to provide Binary format.

* + - 1. ***Type 2: Get\_dummies***

Ocean = pd.get\_dummies(housing[‘Ocean\_proximity’],drop\_first=True)

housing = pd.concat([Ocean,housing],axis=1)

housing.drop([Ocean\_proximity],axis=1,inplace=True)

* + - 1. ***Type 3 : OneHotEncoder***

from sklearn.preprocessing import OneHotEncoder

encoder = OneHotEncoder()

housing\_cat\_1hot = encoder.fit\_transform(housing\_cat\_encoded.reshape(-1,1))

from sklearn.preprocessing import LabelEncoder, OneHotEncoder

labelencoder\_X = LabelEncoder()

X[:, 0] = labelencoder\_X.fit\_transform(X[:, 0])

onehotencoder = OneHotEncoder(categorical\_features = [0])

X = onehotencoder.fit\_transform(X).toarray()

# Encoding the Dependent Variable

labelencoder\_y = LabelEncoder()

y = labelencoder\_y.fit\_transform(y)

***🡪 It will return Sparse Matrix. To Avoid it we can use below Function***

#New Function for One hot Encoder

# Definition of the CategoricalEncoder class, copied from PR #9151.

from sklearn.base import BaseEstimator, TransformerMixin

from sklearn.utils import check\_array

from sklearn.preprocessing import LabelEncoder

from scipy import sparse

class CategoricalEncoder(BaseEstimator, TransformerMixin):

"""Encode categorical features as a numeric array.

The input to this transformer should be a matrix of integers or strings,

denoting the values taken on by categorical (discrete) features.

The features can be encoded using a one-hot aka one-of-K scheme

(``encoding='onehot'``, the default) or converted to ordinal integers

(``encoding='ordinal'``).

This encoding is needed for feeding categorical data to many scikit-learn

estimators, notably linear models and SVMs with the standard kernels.

Read more in the :ref:`User Guide <preprocessing\_categorical\_features>`.

Parameters

----------

encoding : str, 'onehot', 'onehot-dense' or 'ordinal'

The type of encoding to use (default is 'onehot'):

- 'onehot': encode the features using a one-hot aka one-of-K scheme

(or also called 'dummy' encoding). This creates a binary column for

each category and returns a sparse matrix.

- 'onehot-dense': the same as 'onehot' but returns a dense array

instead of a sparse matrix.

- 'ordinal': encode the features as ordinal integers. This results in

a single column of integers (0 to n\_categories - 1) per feature.

categories : 'auto' or a list of lists/arrays of values.

Categories (unique values) per feature:

- 'auto' : Determine categories automatically from the training data.

- list : ``categories[i]`` holds the categories expected in the ith

column. The passed categories are sorted before encoding the data

(used categories can be found in the ``categories\_`` attribute).

dtype : number type, default np.float64

Desired dtype of output.

handle\_unknown : 'error' (default) or 'ignore'

Whether to raise an error or ignore if a unknown categorical feature is

present during transform (default is to raise). When this is parameter

is set to 'ignore' and an unknown category is encountered during

transform, the resulting one-hot encoded columns for this feature

will be all zeros.

Ignoring unknown categories is not supported for

``encoding='ordinal'``.

Attributes

----------

categories\_ : list of arrays

The categories of each feature determined during fitting. When

categories were specified manually, this holds the sorted categories

(in order corresponding with output of `transform`).

Examples

--------

Given a dataset with three features and two samples, we let the encoder

find the maximum value per feature and transform the data to a binary

one-hot encoding.

>>> from sklearn.preprocessing import CategoricalEncoder

>>> enc = CategoricalEncoder(handle\_unknown='ignore')

>>> enc.fit([[0, 0, 3], [1, 1, 0], [0, 2, 1], [1, 0, 2]])

... # doctest: +ELLIPSIS

CategoricalEncoder(categories='auto', dtype=<... 'numpy.float64'>,

encoding='onehot', handle\_unknown='ignore')

>>> enc.transform([[0, 1, 1], [1, 0, 4]]).toarray()

array([[ 1., 0., 0., 1., 0., 0., 1., 0., 0.],

[ 0., 1., 1., 0., 0., 0., 0., 0., 0.]])

See also

--------

sklearn.preprocessing.OneHotEncoder : performs a one-hot encoding of

integer ordinal features. The ``OneHotEncoder assumes`` that input

features take on values in the range ``[0, max(feature)]`` instead of

using the unique values.

sklearn.feature\_extraction.DictVectorizer : performs a one-hot encoding of

dictionary items (also handles string-valued features).

sklearn.feature\_extraction.FeatureHasher : performs an approximate one-hot

encoding of dictionary items or strings.

"""

def \_\_init\_\_(self, encoding='onehot', categories='auto', dtype=np.float64,

handle\_unknown='error'):

self.encoding = encoding

self.categories = categories

self.dtype = dtype

self.handle\_unknown = handle\_unknown

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

"""Fit the CategoricalEncoder to X.

Parameters

----------

X : array-like, shape [n\_samples, n\_feature]

The data to determine the categories of each feature.

Returns

-------

self

"""

if self.encoding not in ['onehot', 'onehot-dense', 'ordinal']:

template = ("encoding should be either 'onehot', 'onehot-dense' "

"or 'ordinal', got %s")

raise ValueError(template % self.handle\_unknown)

if self.handle\_unknown not in ['error', 'ignore']:

template = ("handle\_unknown should be either 'error' or "

"'ignore', got %s")

raise ValueError(template % self.handle\_unknown)

if self.encoding == 'ordinal' and self.handle\_unknown == 'ignore':

raise ValueError("handle\_unknown='ignore' is not supported for"

" encoding='ordinal'")

X = check\_array(X, dtype=np.object, accept\_sparse='csc', copy=True)

n\_samples, n\_features = X.shape

self.\_label\_encoders\_ = [LabelEncoder() for \_ in range(n\_features)]

for i in range(n\_features):

le = self.\_label\_encoders\_[i]

Xi = X[:, i]

if self.categories == 'auto':

le.fit(Xi)

else:

valid\_mask = np.in1d(Xi, self.categories[i])

if not np.all(valid\_mask):

if self.handle\_unknown == 'error':

diff = np.unique(Xi[~valid\_mask])

msg = ("Found unknown categories {0} in column {1}"

" during fit".format(diff, i))

raise ValueError(msg)

le.classes\_ = np.array(np.sort(self.categories[i]))

self.categories\_ = [le.classes\_ for le in self.\_label\_encoders\_]

return self

def transform(self, X):

"""Transform X using one-hot encoding.

Parameters

----------

X : array-like, shape [n\_samples, n\_features]

The data to encode.

Returns

-------

X\_out : sparse matrix or a 2-d array

Transformed input.

"""

X = check\_array(X, accept\_sparse='csc', dtype=np.object, copy=True)

n\_samples, n\_features = X.shape

X\_int = np.zeros\_like(X, dtype=np.int)

X\_mask = np.ones\_like(X, dtype=np.bool)

for i in range(n\_features):

valid\_mask = np.in1d(X[:, i], self.categories\_[i])

if not np.all(valid\_mask):

if self.handle\_unknown == 'error':

diff = np.unique(X[~valid\_mask, i])

msg = ("Found unknown categories {0} in column {1}"

" during transform".format(diff, i))

raise ValueError(msg)

else:

# Set the problematic rows to an acceptable value and

# continue `The rows are marked `X\_mask` and will be

# removed later.

X\_mask[:, i] = valid\_mask

X[:, i][~valid\_mask] = self.categories\_[i][0]

X\_int[:, i] = self.\_label\_encoders\_[i].transform(X[:, i])

if self.encoding == 'ordinal':

return X\_int.astype(self.dtype, copy=False)

mask = X\_mask.ravel()

n\_values = [cats.shape[0] for cats in self.categories\_]

n\_values = np.array([0] + n\_values)

indices = np.cumsum(n\_values)

column\_indices = (X\_int + indices[:-1]).ravel()[mask]

row\_indices = np.repeat(np.arange(n\_samples, dtype=np.int32),

n\_features)[mask]

data = np.ones(n\_samples \* n\_features)[mask]

out = sparse.csc\_matrix((data, (row\_indices, column\_indices)),

shape=(n\_samples, indices[-1]),

dtype=self.dtype).tocsr()

if self.encoding == 'onehot-dense':

return out.toarray()

else:

return out

***# We need to reshape `housing\_cat` to a 2D array:***

cat\_encoder = CategoricalEncoder(encoding="onehot-dense")

housing\_cat\_reshaped = housing\_cat.values.reshape(-1, 1)

housing\_cat\_1hot = cat\_encoder.fit\_transform(housing\_cat\_reshaped)

housing\_cat\_1hot

**🡪 Custom Transformers :** Scikit-Learn Pipeline class helps us in Defining and Executing sequence of transformations such as Custom cleanup operations or Combining specific attributes. Steps:

* + - Create a class
    - Implement three methods

fit()

transform()

fit\_transform()

* ***Create Custom Transformers Class for Combining Attributes***

**#Class to add Experimentnal Columns**

from sklearn.base import BaseEstimator, TransformerMixin

# column index

rooms\_ix, bedrooms\_ix, population\_ix, household\_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, y=None):

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

population\_per\_household = X[:, population\_ix] / X[:, household\_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)

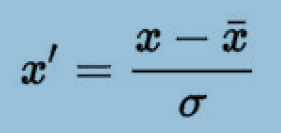
housing\_extra\_attribs = attr\_adder.transform(housing.values)

housing\_extra\_attribs = pd.DataFrame(housing\_extra\_attribs, columns=list(housing.columns)+["rooms\_per\_household", "population\_per\_household"])

housing\_extra\_attribs.head()

* **Feature Scaling :** Min-max Scaling (range b/w 0 & 1) and Standardization
  + **Standardization :** How many standard deviation is the value away from the mean

SKLEARN provides **StandardScaler** Class.

 x = Mean, x- = Current Value

* **Create a class to select numerical or categorical columns**

**# since Scikit-Learn doesn't handle DataFrames yet**

from sklearn.base import BaseEstimator, TransformerMixin

class DataFrameSelector(BaseEstimator, TransformerMixin):

def \_\_init\_\_(self, attribute\_names):

self.attribute\_names = attribute\_names

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

return self

def transform(self, X):

return X[self.attribute\_names].values

* ***Now let's build a pipeline for preprocessing the numerical attributes:***

from sklearn.pipeline import Pipeline

from sklearn.preprocessing import StandardScaler

num\_attr = list(housing\_num)

cat\_attr = ["ocean\_proximity"]

num\_pipeline = Pipeline([

('selector',DataFrameSelector(num\_attr)), **# DataFrame to a NumPy array**

('imputer',Imputer(strategy='median')),

('attr\_adder',CombinedAttributesAdder()),

('scaler',StandardScaler()),

])

cat\_pipeline = Pipeline([

('selector',DataFrameSelector(cat\_attr)),

('cat\_encoder',CategoricalEncoder(encoding="onehot-dense"))

])

#Pipeline Feature Union

from sklearn.pipeline import FeatureUnion

full\_pipeline = FeatureUnion(

transformer\_list=[

("num\_pipeline", num\_pipeline),

("cat\_pipeline", cat\_pipeline),

])

When we call the pipeline’s fit() method

* It calls fit\_transform() sequentially on all transformers
* Passing the output of each call as the parameter to the next transformer
* Until it reaches the final estimator
* For the final estimator it just calls the fit() method

**#Final Prepared Training Set**

housing\_prepared = full\_pipeline.fit\_transform(housing)

1. Explore many different models and short-list the best ones

The goal of this step is to

* Train few(two to five models) and
* Select the best one
* **Linear Regression Model**

**# Train a Linear Regression model**

from sklearn.linear\_model import LinearRegression

lin\_reg = LinearRegression()

lin\_reg.fit(housing\_prepared, housing\_labels)

housing\_pred = lin\_reg.predict(housing\_prepared) **#Prediction Done on Training Set Data**

**# Evaluate the Model Prediction**

from sklearn.metrics import mean\_squared\_error

lin\_mse = mean\_squared\_error(housing\_labels,housing\_pred)

lin\_rmse = np.sqrt(lin\_mse)

**#Underfitting**

* **Decision Tree Regressor Model**

**# Train data with Another Model**

from sklearn.tree import DecisionTreeRegressor

tree\_reg = DecisionTreeRegressor(random\_state=42)

tree\_reg.fit(housing\_prepared, housing\_labels)

housing\_pred = tree\_reg.predict(housing\_prepared)

**# Evaluate the Model Prediction**

from sklearn.metrices import mean\_squared\_error

tree\_mse = mean\_squared\_error(housing\_labels,housing\_pred)

tree\_rmse = np.sqrt(tree\_mse)

**#Overfitting**

* **Since it is overfitting, we need to evaluate using ‘K-fold Cross validation’ :**
* Part of the training set for training
* And part for model validation

# K-fold cross-validation perform randomly splits the training set into K(cv) distinct subsets called folds, then it trains and evaluates the Decision Tree model K times by picking a different fold for evaluation every time and training on the other K-1 folds. The result is an array containing the K evaluation scores.

from sklearn.preprocessing import cross\_val\_score

scores = cross\_cal\_score(tree\_rmse,housing\_prepared,housing\_labels,scoring=’neg\_mean\_square\_error’,cv=10)

tree\_rmse\_scores = np.sqrt(-scores)

**#Function for dislapy the Score of Cross\_validation**

def display\_score(score):

print('Score :',Score)

print('Mean :',score.mean())

print('Standard Deviation :',score.std())

**#Check the Score of Decision Tree**

display\_score(tree\_rmse\_scores)

**#Perform K-fold Cross\_validation on LinearRegression Model**

lin\_reg = LinearRegression()

scores = cross\_val\_score(lin\_reg,housing\_prepared,housing\_labels,scoring=’neg\_mean\_square\_error’,cv=10)

lin\_rmse\_scores = np.sqrt(-scores)

display\_score(lin\_rmse\_scores)

Decision Tree perform worst than Linear Regression in this case because of Overfitting.

* **Random Forest Regressor Model**

**#Train the Model on RandomForest Algorithm**

import sklearn.ensamble import RandomForestRegressor

rand\_reg = RandomForestRegressor(random\_state=42)

rand\_reg.fit(housing\_prepared, housing\_labels)

rand\_pred = rand\_reg.predict(housing\_prepared)

**# Evaluate the Model Prediction**

from sklearn.metrices import mean\_square\_error

rand\_mse = mean\_square\_error(housing\_prepared,housing\_labels)

rand\_rmse = np.sqrt(rand\_mse)

**#Perform K-fold Cross\_validation on LinearRegression Model**

**#conda install py-xgboost (Run in Command Prompt)**

from xgboost import XGBRegressor

xgb = XGBRegressor()

from sklearn.model\_selection import cross\_val\_score

xforest\_scores = cross\_val\_score(xgb, housing\_prepared, housing\_labels,

scoring="neg\_mean\_squared\_error", cv=8)

xforest\_rmse\_scores = np.sqrt(-xforest\_scores)

display\_scores(xforest\_rmse\_scores)

**#Perform K-fold Cross\_validation on LinearRegression Model**

from sklearn.model\_selection import cross\_val\_score

forest\_scores = cross\_val\_score(rand\_reg, housing\_prepared, housing\_labels,

scoring="neg\_mean\_squared\_error", cv=10)

forest\_rmse\_scores = np.sqrt(-forest\_scores)

display\_scores(forest\_rmse\_scores)

1. Fine-tune Model

**One solution is to fiddle with the hyperparameters manually.**

* ***Grid Search*** : Evaluates all possible combinations of hyperparameters values using cross-validation

from sklearn.model\_selection import GridSearchCV

param\_grid = [**# try 12 (3×4) combinations of hyperparameters**

{‘n\_estimators’:[3,10,30],’max\_features’:[2,4,6,8]},

**# then try 6 (2×3) combinations with bootstrap set as False**

{‘bootstrap’:[False],’n\_estimators’:[3,10],’max\_features’:[2,3,4]},]

forest\_reg = RandomForestRegressor(random\_state=42)

grid\_search = GridSearchCV(forest\_reg,param\_grid,cv=5,scoring=’neg\_mean\_square\_error’,n\_jobs=4)

grid\_search.fit(housing\_prepared, housing\_labels)

grid\_search.best\_params\_ **#It will give the best Hyperparameter combinations.**

grid\_search.best\_estimator\_ **#Get the best estimator**

**# Score of each hyperparameter combination tested during the grid search**

cvres = grid\_search.cv\_results\_

for mean\_score,params in zip(cvres["mean\_test\_score"], cvres["params"]):

print(np.sqrt(-mean\_score),params)

**# Show contribution of each attribute/columns towards prediction.**

feature\_importances = grid\_search.best\_estimator\_.feature\_importances\_

extra\_attribs = ["rooms\_per\_hhold", "pop\_per\_hhold", "bedrooms\_per\_room"]

cat\_encoder = cat\_pipeline.named\_steps["cat\_encoder"]

cat\_one\_hot\_attribs = list(cat\_encoder.categories\_[0])

attributes = num\_attr + extra\_attribs + cat\_one\_hot\_attribs

sorted(zip(feature\_importances, attributes), reverse=True)

* ***Randomized Search*** : GridSearchCV is very tedious & almost impossible on large dataset. Instead of trying out all possible combinations, evaluates a given number of random combinations by selecting a random value for each hyperparameter at every iteration

from sklearn.model\_selection import RandomizedSearchCV

from scipy.stats import randint

param\_distribs = {

‘n\_estimators’ : randint(low=1,high=200),

‘max\_features’ : randint(low=1,high=8),

}

forest\_reg = RandomForestRegressor(random\_state=42)

rand\_search = RandomizedSearchCV(forest\_reg,param\_distributions=param\_distribs,n\_iter=1,cv=8,scoring=’neg\_mean\_square\_error’,random\_state=42)

rand\_search.fit(housing\_prepared, housing\_labels)

* ***Another Way is Ensemble Methods.***

1. Evaluate Model on Test Set

**#Finally Make Prediction on Test Set**

***final\_model = grid\_search.best\_estimator\_***

***X\_test = Strat\_test\_set.drop(‘median\_house\_value’,axis=1)***

***Y\_test = Strat\_test\_set[‘median\_house\_value’].copy()***

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

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

***final\_mse = mean\_square\_error(y\_test,final\_prediction)***

***final\_rmse = np.sqrt(final\_rmse)***

1. Present the solution

* *Now we need to present the solution*
  + *What have we learned*
  + *What worked and what did not*
  + *What assumptions were made*
  + *What are model’s limitations*
* **Document everything**
* **Create nice presentations**
  + **With clear visualizations**

1. Launch, monitor, and maintain the system

* *Write monitoring code to check*
  + *Model’s live performance at regular intervals*
  + *Trigger alerts when it drops*
* *Important to catch performance degradation*
  + *As models tends to rot if not trained on fresh data, Evaluate the model’s input data quality from time to time*
* *Maintain the System*
  + *Train model on a regular basis using fresh data*
  + *Automate the process of regularly updating of training model with fresh data*
* *For an online learning system, make sure to save snapshots of its state at regular intervals*