**Notes – Machine Learning**

1. "**pandas**" is the primary tool data scientists use for exploring and manipulating data. The most important part of the Pandas library is the DataFrame. A DataFrame holds the type of data you might think of as a table. This is like a sheet in Excel, or a table in a SQL database.

data = pd.read\_csv(file\_path)

data.describe()

data.columns - outputs name of all the columns in the data

data.dropna()

data.head()

y = data.Column\_Name

1. You can pull out a variable with dot-notation. This single column is stored in a Series, which is broadly like a DataFrame with only a single column of data.
2. "**scikit-learn**" library is used to create your models. When coding, this library is written as sklearn. Scikit-learn is easily the most popular library for modeling the types of data typically stored in DataFrames.
3. **The steps to building and using a model are**:

* Define: What type of model will it be? A decision tree? Some other type of model? Some other parameters of the model type are specified too
* Fit: Capture patterns from provided data. This is the heart of modeling
* Predict: Just what it sounds like
* Evaluate: Determine how accurate the model's predictions are

model = DecisionTreeRegressor(random\_state = 1)

model.fit(observations, ground\_output)

model.predict(test\_data)

1. "**random\_state**" - Specifying a number for random\_state ensures you get the same results in each run. We can use any number, and model quality won't depend meaningfully on exactly what value you choose.
2. "**Model Validation**" - There are many metrics for summarizing model quality, but we'll start with one called Mean Absolute Error (also called MAE). With the MAE metric, we take the absolute value of each error. This converts each error to a positive number. We then take the average of those absolute errors.

mean\_absolute\_error(actual\_y, predicted\_y)

1. "**Splitting the Data**" - The random\_state argument guarantees we get the same split every time

train\_X, test\_X, train\_y, test\_y = train\_test\_split(X, y, random\_state = 0)

1. "**Overfitting**" - When a model matches the training data almost perfectly, but does poorly in validation and other new data

"**Underfitting**" - When a model fails to capture important distinctions and patterns in the data, so it performs poorly even in training data

1. "**Random Forest**" - Decision trees leave you with a difficult decision. A deep tree with lots of leaves will overfit because each prediction is coming from historical data from only the few data at its leaf. But a shallow tree with few leaves will perform poorly because it fails to capture as many distinctions in the raw data.

Even today's most sophisticated modeling techniques face this tension between underfitting and overfitting. But, many models have clever ideas that can lead to better performance. One of the such model is - "Random Forest".

The random forest uses many trees, and it makes a prediction by averaging the predictions of each component tree. It generally has much better predictive accuracy than a single decision tree and it works well with default parameters.

model = RandomForestRegressor(random\_state=1)

1. Python libraries represent missing numbers as nan which is short for "not a number". You can detect which cells have missing values, and then count how many there are in each column with the command:

missing\_val\_count\_by\_column = (data.isnull().sum())

print(missing\_val\_count\_by\_column[missing\_val\_count\_by\_column > 0])

1. There are many ways data can end up with missing values. For example:

* A 2-bedroom house wouldn't include an answer for "How large is the third bedroom?"
* Someone being surveyed may choose not to share their income

1. 3 ways for handling missing data:

* A Simple Option: Drop Columns with Missing Values - Same columns need to be dropped from training and test data. Generally, not a good approach.

cols\_with\_missing = [col for col in original\_data.columns

if original\_data[col].isnull().any()]

redued\_original\_data = original\_data.drop(cols\_with\_missing, axis=1)

reduced\_test\_data = test\_data.drop(cols\_with\_missing, axis=1)

* A Better Option: Imputation - The default behavior fills in the mean value for imputation.

from sklearn.impute import SimpleImputer

my\_imputer = SimpleImputer()

data\_with\_imputed\_values = my\_imputer.fit\_transform(original\_data)

* An Extension to Imputation

Q) How to retain column headers of data frame after pre-processing in scikit-learn? It usually get converted into numbers.

Ans) scikit-learn indeed strips the column headers in most cases, so just add them back on afterward.

X\_imputed\_df = pd.DataFrame(X\_imputed, columns = X\_train.columns)

1. List Comprehension in Python

<https://www.pythonforbeginners.com/basics/list-comprehensions-in-python>

<https://www.programiz.com/python-programming/list-comprehension>

new\_list = [expression(variable) for variable in data if filter(variable)]

Q) What all is considered as missing values?

Ans)

|  |  |
| --- | --- |
| **Missing Values** | **Not Missing Values** |
| Blank (empty cell) | Nan |
| NaN | NAN |
| nan |  |

Q) What is the difference between **isna()** and **isnull()** in python ?

Ans) <https://datascience.stackexchange.com/questions/37878/difference-between-isna-and-isnull-in-pandas>

Q) What do these different functions do?

Ans) **data.isnull()** – Prints Boolean for each cell of the data, if cell contains missing value or not

**data.isnull().sum()** – Print a series type of integer values, of total number of missing values in each column

**data.isnull().any()** – Print a series type of boolean values, of whether column contains any missing values or not

**data.dropna(axis = 1)** – Removes all the columns with the missing values (can be used for rows also)

**data.drop(cols\_with\_missing\_data, axis = 1)** – Removes all the columns with the missing values (can be used for rows also)

Q) Understand sklearn.impute.SimpleImputer?

Ans) <http://scikit-learn.org/dev/modules/generated/sklearn.impute.SimpleImputer.html>

SimpleImputer(*missing\_values=nan*, *strategy=”mean”*, *fill\_value=None*, *verbose=0*, *copy=True*)

**strategy : string, optional (default=”mean”)**

* If “**mean**”, then replace missing values using the mean along each column. Can only be used with numeric data.
* If “**median**”, then replace missing values using the median along each column. Can only be used with numeric data.
* If “**most\_frequent**”, then replace missing using the most frequent value along each column. Can be used with strings or numeric data.
* If “**constant**”, then replace missing values with fill\_value. Can be used with strings or numeric data.

1. Categorical data is data that takes only a limited number of values. You will get an error if you try to plug these variables into most machine learning models in Python without "encoding" them first.

**One hot encoding** is the most widespread approach, and it works very well unless your categorical variable takes on many values (i.e. you generally won't it for variables taking more than 15 different values. It'd be a poor choice in some cases with fewer values, though that varies.)

One hot encoding creates new (binary) columns, indicating the presence of each possible value from the original data. How naming of new columns done is:

OriginalColumnName\_EncodedValue

Pandas assigns a data type or attribute (called a dtype) to each column in the DataFrame or Series (not to the list type). "**Object**" indicates a column has text (there are other things it could be theoretically be, but that's unimportant for our purposes). It's most common to one-hot encode these "object" columns, since they can't be plugged directly into most models.

print(data.dtypes)

predictors\_without\_categoricals = data.select\_dtypes(exclude=[“Object”])

predictors\_without\_categoricals = data.select\_dtypes(include=[“Object”]) – To get only Object type columns

Pandas offers a convenient function called **get\_dummies** to get one-hot encodings. It can be passed both numeric and categorical data, but it will encode only “Object” type data.

one\_hot\_encoded\_training\_predictors = pd.get\_dummies(train\_predictors)

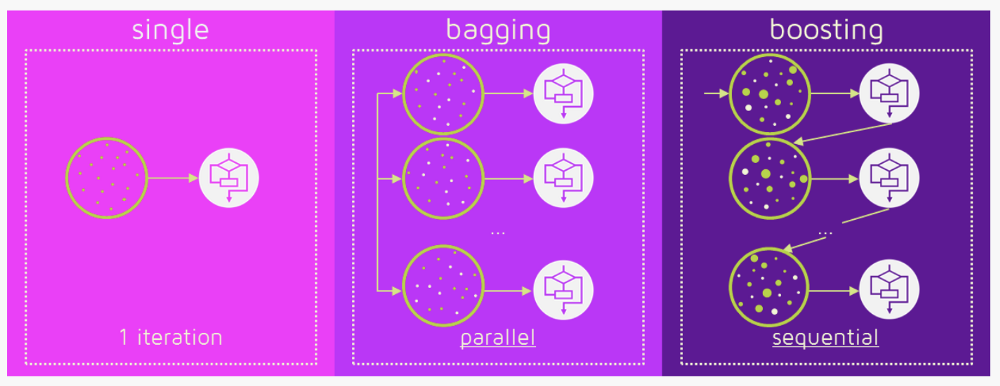
**Remember:** One-hot encoding does not encode missing values (like NaN or nan or blank cell), just ignores these values and encode the rest legitimate entries. It will show all 0’s for a cell having missing value.

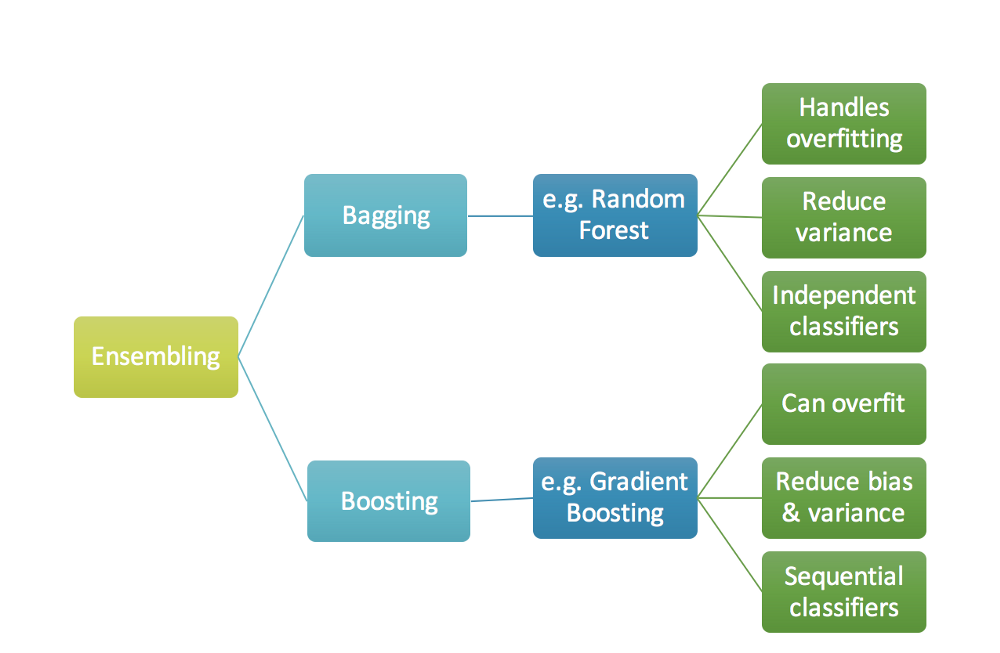
1. When we try to predict the target variable using any machine learning technique, the main causes of difference in actual and predicted values are noise, variance, and bias. Ensemble helps to reduce these factors (except noise, which is irreducible error).

In machine learning, **Ensemble** means combining diverse set of learners (individual models) together to improvise on the stability and predictive power of the final model.

Ensembling techniques are further classified into Bagging and Boosting:

* **Bagging** - Bagging is a simple ensembling technique in which we build many independent predictors/models/learners and combine them using some model averaging techniques. (e.g. weighted average, majority vote or normal average). E.g.: Random Forest
* **Boosting** - Boosting is an ensemble technique in which the predictors are not made independently, but sequentially. The predictors can be chosen from a range of models like decision trees, regressors, classifiers etc. Because new predictors are learning from mistakes committed by previous predictors, it takes less time/iterations to reach close to actual predictions. But we must choose the stopping criteria carefully or it could lead to overfitting on training data. Gradient Boosting is an example of boosting algorithm.





1. **Random Forest** - Random forests are a supervised ensemble learning method for classification, regression and other tasks, that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode (occurring maximum times) of the classes (classification) or mean prediction (regression) of the individual trees. Random forests correct decision trees habit of overfitting to their training set.

To address why random forest algorithm. I am giving you the below advantages:

* The same random forest algorithm can be used for both classification and regression task.
* Random forest classifier will handle the missing values.
* When we have more trees in the forest, random forest classifier won’t overfit the model.
* Can model the random forest classifier for categorical values also.

Q) Explain **data.dropna()** function?

Ans) The signature of the function is:

data.dropna(*axis=0*, *how='any'*, *thresh=None*, *subset=None*, *inplace=False*)

* **axis** : {0 or ‘index’, 1 or ‘columns’}, default 0 – Where 0 is for rows and 1 for columns
* **subset** : array-like, optional – Check for missing values only in this subset. If missing values found in any of the subset columns, then the entire row is deleted. If no missing value found in the entire subset, row is not deleted even if missing values are present in other columns of same row.
* **inplace** : bool, default False – If True, then no need to re-assign modified data and changes are made to the same variable. If False, then we need to re-assign to same or different variable as per our wish.

1. **XGBoost** is the leading model for working with standard tabular data (the type of data you store in Pandas DataFrames, as opposed to more exotic types of data like images and videos). XGBoost is an implementation of the Gradient Boosted Decision Trees algorithm.

Gradient boosting is a machine learning technique for regression and classification problems, which produces a prediction model in the form of an ensemble of weak prediction models, typically decision trees.

We go through cycles that repeatedly builds new models and combines them into an ensemble model. To make a prediction, we add the predictions from all previous models. We can use these predictions to calculate new errors, build the next model, and add it to the ensemble.

my\_model = XGBRegressor()

my\_model.fit(train\_X, train\_Y, verbose = False)

predictions = my\_model.predict(test\_X)

print(mean\_absolute\_error(test\_Y, predictions))

1. **XGBoost Regressor Model Tuning Parameters**:

* **n\_estimators** – It specifies how many times to go through the modeling cycle of the algorithm.
* **early\_stopping\_rounds** – Too high value for n\_estimators usually cause overfitting. The argument early\_stopping\_rounds offers a way to automatically find the ideal value.

Early stopping causes the model to stop iterating when the validation score stops improving, even if we aren't at the hard stop for n\_estimators. It's smart to set a high value for n\_estimators and then use early\_stopping\_rounds to find the optimal time to stop iterating.

Through early\_stopping\_rounds we specify number of rounds of straight deterioration to allow before stopping.

When using early\_stopping\_rounds, you need to set aside some of your data for checking the number of rounds to use. If you later want to fit a model with all your data, set n\_estimators to whatever value you found to be optimal when run with early stopping.

XGBoost supports a suite of evaluation metrics not limited to:

* + - * 1. “*rmse*” for root mean squared error
        2. “*mae*” for mean absolute error
        3. “*logloss*” for binary logarithmic loss and “mlogloss” for multi-class log loss (cross entropy)
        4. “*error*” for classification error
        5. “*auc*” for area under ROC curve
* **learning\_rate**
* **n\_jobs** - On larger datasets where runtime is a consideration, you can use parallelism to build your models faster. It's common to set the parameter n\_jobs equal to the number of cores on your machine. On smaller datasets, this won't help.

my\_model = XGBRegressor(n\_estimators=1000, learning\_rate=0.05)

my\_model.fit(train\_X, train\_Y, early\_stopping\_rounds=5, eval\_set=[(val\_X, val\_Y)], eval\_metric = “rmse”, verbose = False)

1. **Pipeline** - Pipelines are a simple way to keep your data processing and modeling code organized. Specifically, a pipeline bundles preprocessing and modeling steps, so you can use the whole bundle as if it were a single step. Benefits are as follows:

* **Cleaner Code**: You won't need to keep track of your training (and validation) data at each step of processing. Accounting for data at each step of processing can get messy. With a pipeline, you don't need to manually keep track of each step.
* **Fewer Bugs**: There are fewer opportunities to mis-apply a step or forget a pre-processing step.
* **Easier to Productionize**: It can be surprisingly hard to transition a model from a prototype to something deployable at scale. We won't go into the many related concerns here, but pipelines can help.
* **More Options for Model Testing**: You will see an example in the next tutorial, which covers cross-validation.

my\_pipeline = make\_pipeline(SimpleImputer(), RandomForestRegressor())

my\_pipeline.fit(train\_X, train\_Y)

my\_pipeline\_predictions = my\_pipeline.predict(val\_X)

print(mean\_absolute\_error(val\_Y, my\_pipeline\_predictions)

1. Most scikit-learn objects are either **transformers** or **models**.

**Transformers** are for pre-processing before modeling. The Imputer class (for filling in missing values) is an example of a transformer. We can use multiple transformers sequentially.

**Models** are used to make predictions. You will usually preprocess your data (with transformers) before putting it in a model.

You can tell if an object is a transformer or a model by how you apply it. After fitting a transformer, you apply it with the transform command. After fitting a model, you apply it with the predict command. Your pipeline must start with transformer steps and end with a model.

Q) Difference between fit(), transform(), fit\_transform() and predict() in scikit-learn ?

Ans) It depends on which object we are using these methods:

**For Transformers:**

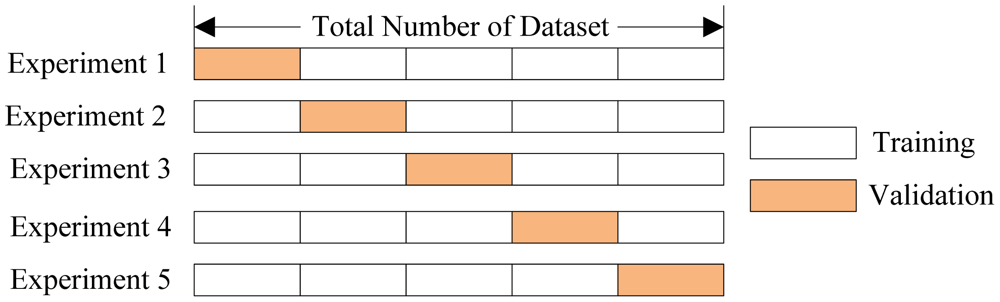
* fit() - It is used for calculating the initial filling of parameters on the training data (like mean of the column values) and saves them as an internal objects state
* transform() - Use the above calculated value and return modified training data. **We cannot call transform(), without first calling fit(). This is independent of data. Means we can do this – first call fit() on training data, then transform() on training data and then directly call transform() on test data (because the parameters are already stored in object state).**
* fit\_transform() - It just joins above two steps. Internally, it just calls first fit() and then transform() on the same data.

**For Models:**

* fit() - It just calculates the parameters/weights on training data (e.g. parameters returned by coef() in case of Linear Regression) and saves them as an internal objects state.
* predict() - Use the above calculated weights on test data to make the predictions
* transform() - Cannot be used
* fit\_transform() - Cannot be used

1. Cross Validation – There are a few shortcomings with Train\_Test\_Split approach which are mentioned below. To overcome these we use **Cross Validation**.

* Limited data can lead to very less training and testing data
* We test our model only on a single set, which cannot everytime give accurate measure of amodel quality



my\_pipeline = make\_pipeline(RandomForestRegressor())

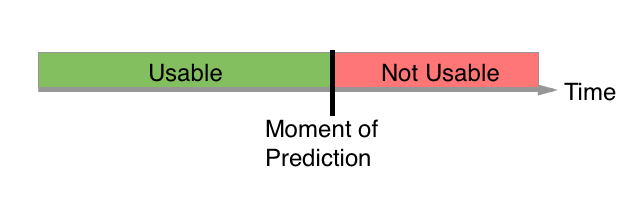
scores = cross\_val\_score(my\_pipeline, X, y, scoring = “neg\_mean\_absolute\_error”) – Remember here X and y are complete training data and not divided into training and validation data.

avg\_score = -1 \* scores.mean()

1. There are two main types of leakage: **Leaky Predictors** and a **Leaky Validation Strategies.**

**Leaky Predictors**

This occurs when your predictors include data that will not be available at the time you make predictions. To prevent this type of data leakage, any variable updated (or created) after the target value is realized should be excluded. Because when we use this model to make new predictions, that data won't be available to the model.



**Leaky Validation Strategy**

A much different type of leak occurs when you aren't careful distinguishing training data from validation data. For example, this happens if you run preprocessing (like fitting the Imputer for missing values) before calling train\_test\_split. Validation is meant to be a measure of how the model does on data it hasn't considered before. You can corrupt this process in subtle ways if the validation data affects the preprocessing behavoir.. The end result? Your model will get very good validation scores, giving you great confidence in it, but perform poorly when you deploy it to make decisions.

**Preventing Leaky Validation Strategies**

If your validation is based on a simple train-test split, exclude the validation data from any type of *fitting*, including the fitting of preprocessing steps. This is easier if you use [scikit-learn Pipelines](https://www.kaggle.com/dansbecker/pipelines). When using cross-validation, it's even more critical that you use pipelines and do your preprocessing inside the pipeline.

**Notes – Pandas**

1. Syntax to create a DataFrame:

pd.DataFrame(index = ['2017 Sales', '2018 Sales'], columns = ['Apples', 'Bananas'], data = [[35, 21], [41, 34]]) – Data is feeded row-wise

OR

pd.DataFrame(index = ['2017 Sales', '2018 Sales'], data = {'Apples': [35, 41], 'Bananas': [21, 34]}) – Data is feeded column-wise

1. Syntax to create a Series: (**I think Series is a Row-Vector**)

pd.Series(index = [‘2017 Sales’, ‘2018 Sales’, ‘2019 Sales’], data = [‘Apples’, ‘Bananas’, ‘Grapes’], name = ‘Dinner’)

1. For selecting a column from DataFrame:

dataframe\_name.column\_name – Select complete column

dataframe\_name.column\_name[3] – Return 4th value from dataframe of particular column

For selecting rows from DataFrame:

dataframe\_name.iloc[4] – Return 5th row from dataframe

**Notes – Data Visualization**

Q) Difference between Categorical(Nominal) variable, Ordinal variable and Interval data?

Ans) The difference is as follows:

**Nominal Values**: "pure" categories that don't make a lot of sense to order. These are the variables that just state a fact. Nominal categorical variables include things like countries, ZIP codes, types of cheese, and gender.

reviews[“points”].value\_counts().head(10).plot.bar() – For actual counts

(reviews[“points”].value\_counts().head(10) / len(reviews)).plot.bar() – For percentages

**Ordinal Values**: Things that do make sense to compare, like earthquake magnitudes, marks.

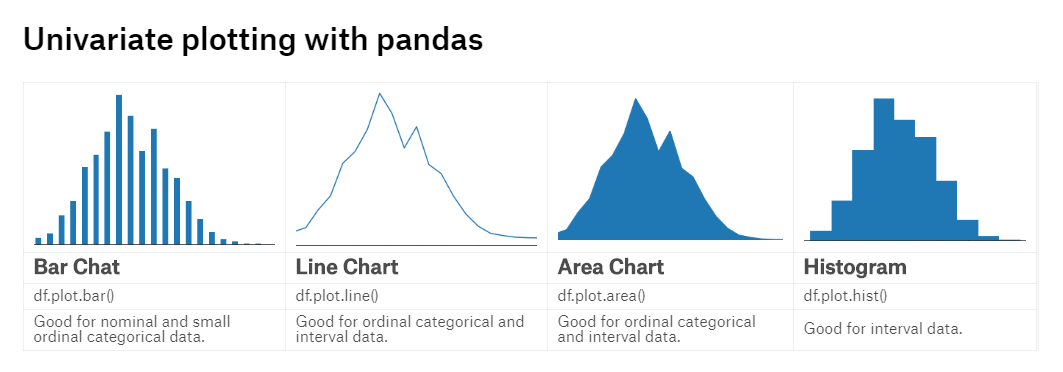
reviews[“points”].value\_counts().sort\_index().plot.line()

reviews[“points”].value\_counts().sort\_index().plot.area()

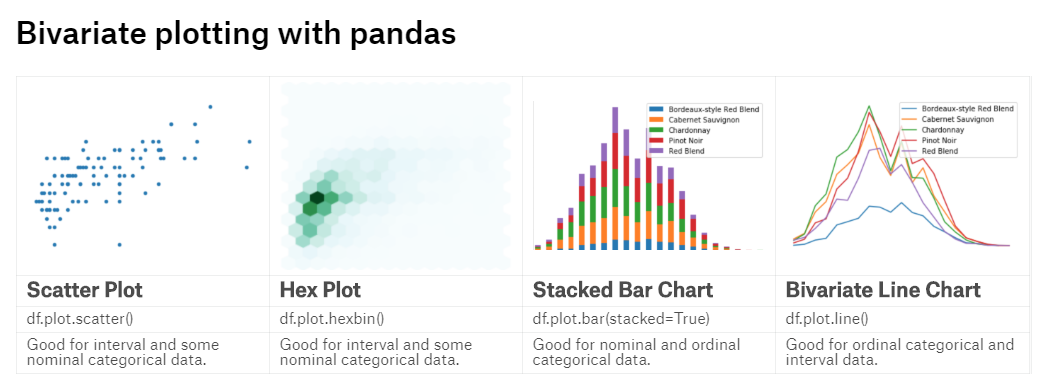
**Interval Data**: An interval variable goes beyond an ordinal variable: it has a *meaningful* order, in the sense that we can quantify what the difference between two entries is itself an interval variable. Eg: Temperature

reviews[reviews[“price”] < 200][“price”].plot.hist()

reviews[“price”].plot.hist()



Q) What does **Series.value\_counts()** function do ?

Ans) Returns object containing counts of unique values. The resulting object will be in descending order so that the first element is the most frequently-occurring element. Excludes NA values by default.  
  


**Scatter Plot**: Because of their weakness to overplotting, scatter plots work best with relatively small datasets, and with variables which have a large number of unique values.

training\_data.sample(100).plot.scatter(x=’YearBuilt’, y=’TotRmsAbvGrd’)

**Hex Plot**: This plot overcome the overplotting weakness of the scatter plot and shows values at same points through different intensity levels. Higher the intensity means more values are concentrated at that point.

training\_data.plot.hexbin(x='YearBuilt', y='TotRmsAbvGrd', gridsize = 15)

**Stacked Bar Charts**: This is used to plot multiple variables one top on the other, such that we have single variable on x-axis and multiple variables on y-axis.

features\_training\_data.sample(100).plot.bar(stacked=True, x='YearBuilt', y=['FullBath', 'BedroomAbvGr', 'TotRmsAbvGrd'])

**Bivariate Line Chart**: This is used to plot multiple variables in a more presentable format, such that we have single variable on x-axis and multiple variables on y-axis.

features\_training\_data.sample(100).plot.line(x='YearBuilt', y=['FullBath', 'BedroomAbvGr', 'TotRmsAbvGrd'])

1. Under the hood, pandas data visualization tools are built on top of another, lower-level graphics library called matplotlib. Anything that you build in pandas can be built using matplotlib directly. Pandas merely make it easier to get that work done.

matplotlib *does* provide a way of adjusting the title size.

graph = features\_training\_data["YearBuilt"].value\_counts().plot.bar(figsize=(18,12), fontsize=15)

graph.set\_title("Year plot of the different houses", fontsize = 20)

1. seaborn *also* uses matplotlib under the hood. This means that the tricks above work there too. seaborn has its own tricks, too.

import seaborn as sns

sns.despine(bottom=True, left=True) – Removes black axes from bottom and left

1. **Subplotting**: It is a technique for creating multiple plots that live side-by-side in one overall figure. We can use the subplots method to create a figure with multiple subplots. subplots takes two arguments. The first one controls the number of *rows*, the second one the number of *columns*.

import matplotlib.pyplot as plt

fig, axarr = plt.subplots(2, 3, figsize=(12,8))

subplots returns two things, a figure (which we assigned to fig) and an array of the axes contained therein (which we assigned to axarr). To tell pandas which subplot we want a new plot to go in—the first one or the second one—we need to grab the proper axis out of the list and pass it into pandas via the ax parameter:

reviews[“points”].value\_counts().head(10).plot.bar(ax=axarr[0][2], fontsize=12, color=”red”)

axarr[0][2].set\_title(“Points of Wine”, fontsize=15)

subplots are critically useful because they enable **faceting**. Faceting is the act of breaking data variables up across multiple subplots, and combining those subplots into a single figure. So instead of one bar chart, we might have, say, four, arranged together in a grid.

However, the recommended way to perform faceting is to use the seaborn FacetGrid facility.

