Concepts to be understood:

* Standard deviation
* Root mean square

In Andrew NG:

Start from starting

Avoid octave/matlab

Continue till regularaization

Which topic is best in which book:

* Applying Minmax scaler and grid search using pipelines is best explained in <https://learning.oreilly.com/library/view/introduction-to-machine/9781449369880/>

POINTS TO BE NOTED:

* ML definitions:
* Arthur samuel def: ML is a field of study that gives computers the ability to learn without being explicitly programmed
* Tom Mitchell def: A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E(my own version: a computer is said to be learning from an experience E from a task T with experience E if, the Performance on T improves with Experience)
* In simple words performance should be directly proportional to experience of an task
* A **quadratic function** is one of the form **f(x) = ax2 + bx + c**, where **a**,**b**, and **c** are numbers with **a** not equal to zero.

An example:

Diagram

Description automatically generated

* second degree polynomial equation example **x2-44x+ 435 = 0**.
* Difference between classification and regression:

**what you are trying to model or predict Classification you are trying to predict a label.**

* Is this red or blue?
* Is it left or right?
* It is true or false?
* Is it a bird, yes or no?
* Is it north, south, east or west?

Any question that can be answered with a label like these above is a classification.

It may be that you use 0 and 1 as target proxies, but the value you are trying to predict is not 0 or 1 it is the label represented by 1 ..is this red 1 yes, 0 no.

**Regression you are trying to predict an actual numerical value**

(typically continuous with any shade of decimal values between but there are legitimate regression with integers two).

* What is the temperature outside?
* How many milligrams does this quail egg weigh?
* How much corn will be produce on this acre of land?
* How many seconds until the next eruption at Old Faithful? (this is an integer version)
* ML is mainly classified (there are other classifications as well):
* Supervised learning (most used):
* (Informal definition): right answers(labels) are already given, and the machine has to predict more right answers
* Supervised learning problems are categorized into "**regression**" and "**classification**" problems
* Regression will deal with continuous data, classification will deal with discrete data

Example:

* Given data about the size of houses on the real estate market, try to predict their price. Price as a function of size is a continuous output, so this is a regression problem.
* We could turn this example into a classification problem by instead making our output about whether the house "sells for more or less than the asking price." Here we are classifying the houses based on price into two discrete categories.
* Unsupervised Learning:
* Types of variables:
* Continuous variable: if a variable can take any value between its maximum and minimum range
* Discrete variable: if a variable is countable in finite amount of time. (you can measure it)

If there is a race, number of people in race is discrete, the time taken by an individual to complete a race is continuous variable

* In a regression problem, our goal is to predict a continuous valued output
* Ex: time taken by an athlete to complete marathon, when we have some features such as age, length of marathon, previous track record
* A classification problem will have a discrete valued output of 0 or 1,sometimes there are more than 2 possible outputs
* Ex: if you are classifying the price of house, you can classify them in to, extremely priced (as 0), moderately priced(as 1), affordable(as 2)
* Working with preprocessed attributes is not a problem and it is common in ML, you should try to understand how the data was computed
* Cost function: cost function is used to measure how wrong the model is in determining the relation between input and output. In less words it tells how badly the model is predicting or behaving
* What is utility function?
* **Confidence interval**- it is the way of quantifying the uncertainty of an estimate.
* When working with preprocessed data, if the target variable is capped or forcedly kept in the range because the people who collected data were not able to collect correct values, because of some limitations, it is your duty now to check with the client to see if they want precise values(because your ML algorithm may learn that it will never have the output beyond the capped range), if they want precise values, then
  + Remove the end value(rows) which are capped
  + Or try to get the actual values of such cases
* Before you start doing some visualizations on the data, it is better to keep some data aside for testing purposes
  + Reason: It may sound strange to voluntarily set aside part of the data at this stage. After all, you have only taken a quick glance at the data, and surely you should learn a whole lot more about it before you decide what algorithms to use, right? This is true, but your brain is an amazing pattern detection system, which means that it is highly prone to overfitting: if you look at the test set, you may stumble upon some seemingly interesting pattern in the test data that leads you to select a particular kind of Machine Learning model. When you estimate the generalization error using the test set, your estimate will be too optimistic, and you will launch a system that will not perform as well as expected. This is called **data snooping bias**.
* Creating a test set is theoretically simple: pick some instances randomly, typically 20% of the dataset (or less if your dataset is very large), and set them aside
* Scikit-Learn provides a few functions to split datasets into multiple subsets in various ways. The simplest function is train\_test\_split(), which does pretty much the same thing as the function split\_train\_test(), with a couple of additional features. First, there is a random\_state parameter that allows you to set the random generator seed. Second, you can pass it multiple datasets with an identical number of rows, and it will split them on the same indices (this is very useful, for example, if you have a separate DataFrame for labels):

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

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

* stratified sampling: So far we have considered purely random sampling methods. This is generally fine if your dataset is large enough (especially relative to the number of attributes), but if it is not, you run the risk of introducing a significant sampling bias. When a survey company decides to call 1,000 people to ask them a few questions, they don’t just pick 1,000 people randomly in a phone book. They try to ensure that these 1,000 people are representative of the whole population. For example, the US population is 51.3% females and 48.7% males, so a well-conducted survey in the US would try to maintain this ratio in the sample: 513 female and 487 male. This is called **stratified sampling**: the population is divided into homogeneous subgroups called strata, and the right number of instances are sampled from each stratum to guarantee that the test set is representative of the overall population. If the people running the survey used purely random sampling, there would be about a 12% chance of sampling a skewed test set that was either less than 49% female or more than 54% female. Either way, the survey results would be significantly biased.
* If you know that a particular column is contributing more importance to the target prediction then we may want to make sure that particular data is having stratified split in of equal importance in both train and test data, pfb the code

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(housing, housing["income\_cat"]):

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

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

* Another way to check for correlation between attributes is to use the pandas scatter\_matrix() function, which plots every numerical attribute against every other numerical attribute, you can also use a list of attributes only which you want by passing them

scatter\_matrix(housing["median\_house\_value", "median\_income", "total\_rooms",

"housing\_median\_age"], figsize=(12, 8))

* Most Machine Learning algorithms cannot work with missing features, so let’s create a few functions to take care of them., so let’s fix this. You have three options:

1. Get rid of the corresponding districts.
2. Get rid of the whole attribute.
3. Set the values to some value (zero, the mean, the median, etc.).

You can accomplish these easily using DataFrame’s dropna(), drop(), and fillna() methods:

* Scikit-Learn provides a handy class to take care of missing values: SimpleImputer. Here is how to use it. First, you need to create a SimpleImputer instance, specifying that you want to replace each attribute’s missing values with the median of that attribute:

from sklearn.impute import SimpleImputer

imputer = SimpleImputer(strategy="median")

imputer.fit(<dataframe>)

* We cannot be sure that only particular column will have empty values, so it is a good practice to apply imputer to all columns
* he imputer has simply computed the median of each attribute and stored the result in its statistics\_ instance variable.

Text

Description automatically generated with medium confidence

* Now you can use this “trained” imputer to transform the training set by replacing missing values with the learned medians:

X = imputer.transform(housing\_num)

The result is a plain NumPy array containing the transformed features. If you want to put it back into a pandas DataFrame, it’s simple:

housing\_tr = pd.DataFrame(X, columns=housing\_num.columns,

index=housing\_num.index

* You can notice that in the above example, how the columns and index parameters have been used to assign columns and index values to a numpy array X to convert it in to a dataframe
* Estimators

Any object that can estimate some parameters based on a dataset is called an ***estimator*** (e.g., an imputer is an estimator). The estimation itself is performed by the fit() method, and it takes only a dataset as a parameter (or two for supervised learning algorithms; the second dataset contains the labels). Any other parameter needed to guide the estimation process is considered a hyperparameter (such as an imputer’s strategy), and it must be set as an instance variable (generally via a constructor parameter).

* Transformers

Some estimators (such as an imputer) can also transform a dataset; these are called ***transformers***. Once again, the API is simple: the transformation is performed by the transform() method with the dataset to transform as a parameter. It returns the transformed dataset. This transformation generally relies on the learned parameters, as is the case for an imputer. All transformers also have a convenience method called fit\_transform() that is equivalent to calling fit() and then transform() (but sometimes fit\_transform() is optimized and runs much faster).

* Predictors

Finally, some estimators, given a dataset, are capable of making predictions; they are called ***predictors***. For example, the LinearRegression model in the previous chapter was a predictor: given a country’s GDP per capita, it predicted life satisfaction. A predictor has a **predict**() method that takes a dataset of new instances and returns a dataset of corresponding predictions. It also has a **score**() method that measures the quality of the predictions, given a test set (and the corresponding labels, in the case of supervised learning algorithms)

* Inspection

All the estimator’s hyperparameters are accessible directly via public instance variables (e.g., **imputer.strategy**), and all the estimator’s learned parameters are accessible via public instance variables with an underscore suffix (e.g., **imputer.statistics**\_).

* Nonproliferation of classes

Datasets are represented as NumPy arrays or SciPy sparse matrices, instead of homemade classes. Hyperparameters are just regular Python strings or numbers.

* Composition

Existing building blocks are reused as much as possible. For example, it is easy to create a Pipeline estimator from an arbitrary sequence of transformers followed by a final estimator, as we will see.

* Sensible defaults

Scikit-Learn provides reasonable default values for most parameters, making it easy to quickly create a baseline working system.

* As most of ML algorithms prefer to work with numbers, you need to find a way to transform text to number(if the text values are repeatable (like a set of values which repeat)), it is done through encoding using a module available in sklearn

from sklearn.preprocessing import OrdinalEncoder

>>> ordinal\_encoder = OrdinalEncoder()

>>> housing\_cat\_encoded = ordinal\_encoder.fit\_transform(housing\_cat)

>>> housing\_cat\_encoded[:3]

array([[0.],

[0.],

[4.])

To see the list of values that are repeated use the below code:

ordinal\_encoder.categories\_

[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY'],

dtype=object)]

One issue with this representation is that ML algorithms will assume that two nearby values are more similar than two distant values. This may be fine in some cases (e.g., for ordered categories such as “bad,” “average,” “good,” and “excellent”), but it is obviously not the case for the ocean\_proximity column (for example, categories 0 and 4 are clearly more similar than categories 0 and 1). To fix this issue, a common solution is to create one binary attribute per category: one attribute equal to 1 when the category is “<1H OCEAN” (and 0 otherwise), another attribute equal to 1 when the category is “INLAND” (and 0 otherwise), and so on. This is called ***one-hot encoding***, because only one attribute will be equal to 1 (hot), while the others will be 0 (cold). The new attributes are sometimes called *dummy* attributes. Scikit-Learn provides a OneHotEncoder class to convert categorical values into one-hot vectors:[**20**](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch02.html#idm45022182773944)

>>> from sklearn.preprocessing import OneHotEncoder

>>> cat\_encoder = OneHotEncoder()

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

>>> housing\_cat\_1hot

<16512x5 sparse matrix of type '<class 'numpy.float64'>'

with 16512 stored elements in Compressed Sparse Row format>

Notice that the output is a SciPy *sparse matrix*, instead of a NumPy array. This is very useful when you have categorical attributes with thousands of categories. After one-hot encoding, we get a matrix with thousands of columns, and the matrix is full of 0s except for a single 1 per row. Using up tons of memory mostly to store zeros would be very wasteful, so instead a sparse matrix only stores the location of the nonzero elements. You can use it mostly like a normal 2D array,[**21**](https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/ch02.html#idm45022182722376) but if you really want to convert it to a (dense) NumPy array, just call the toarray() method:

>>> housing\_cat\_1hot.toarray()

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

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

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

...,

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

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

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

Once again, you can get the list of categories using the encoder’s **categories\_**  instance variable:

>>> cat\_encoder.categories\_

[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],

dtype=object)]

**Feature Scaling**

One of the most important transformations you need to apply to your data is *feature scaling*. With few exceptions, Machine Learning algorithms don’t perform well when the input numerical attributes have very different scales. This is the case for the housing data: the total number of rooms ranges from about 6 to 39,320, while the median incomes only range from 0 to 15. Note that scaling the target values is generally not required.

There are two common ways to get all attributes to have the same scale: *min-max scaling* and *standardization*.

Min-max scaling (many people call this *normalization*) is the simplest: values are shifted and rescaled so that they end up ranging from 0 to 1. We do this by subtracting the min value and dividing by the max minus the min. Scikit-Learn provides a transformer called MinMaxScaler for this. It has a feature\_range hyperparameter that lets you change the range if, for some reason, you don’t want 0–1.

**Standard deviation(std dev) and mean have same units, as variance is having square of std dev the unit of variance is the square of unit of std dev**

Standardization is different: first it subtracts the mean value (so standardized values always have a zero mean), and then it divides by the standard deviation so that the resulting distribution has unit variance. Unlike min-max scaling, standardization does not bound values to a specific range, which may be a problem for some algorithms (e.g., neural networks often expect an input value ranging from 0 to 1). **However, standardization is much less affected by outliers. For example, suppose a district had a median income equal to 100 (by mistake). Min-max scaling would then crush all the other values from 0–15 down to 0–0.15, whereas standardization would not be much affected. Scikit-Learn provides a transformer called StandardScaler for standardization.**

**WARNING**

As with all the transformations, it is important to fit the scalers to the training data only, not to the full dataset (including the test set). Only then can you use them to transform the training set and the test set (and new data).

* Custom Transformers

**Although Scikit-Learn provides many useful transformers, you will need to write your own for tasks such as custom cleanup operations or combining specific attributes**. **You will want your transformer to work seamlessly with Scikit-Learn functionalities (such as pipelines), and since Scikit-Learn relies on duck typing (not inheritance), all you need to do is create a class and implement three methods: fit() (returning self), transform(), and fit\_transform().**

**You can get the last one for free by simply adding TransformerMixin as a base class. If you add BaseEstimator as a base class (and avoid \*args and \*\*kargs in your constructor), you will also get two extra methods (get\_params() and set\_params()) that will be useful for automatic hyperparameter tuning.**

(refer Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition for more information)

* For detailed explanation on pipeline builder and column transformer you can refer to (Transformation pipelines and Feature Scaling in chapter 2 of Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition for more information)

When a ml model is underfitting we turn to complex model and if it overfits (or the vice versa), then it is needed use validation methods such as cross fold validation, the process is to evaluate the Decision Tree model would be to use the train\_test\_split() function to split the training set into a smaller training set and a validation set, then train your models against the smaller training set and evaluate them against the validation set. It’s a bit of work, but nothing too difficult, and it would work fairly well.

A great alternative is to use Scikit-Learn’s *K-fold cross-validation* feature. The following code randomly splits the training set into 10 distinct subsets called *folds*, then it trains and evaluates the Decision Tree model 10 times, picking a different fold for evaluation every time and training on the other 9 folds. The result is an array containing the 10 evaluation scores:

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

scores = cross\_val\_score(tree\_reg, housing\_prepared, housing\_labels,

scoring="neg\_mean\_squared\_error", cv=10)#here I think, cv(cross validation) means 10 the 10 partitions we discussed above

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

###### WARNING

Scikit-Learn’s cross-validation features expect a utility function (greater is better) rather than a cost function (lower is better), so the scoring function is actually the opposite of the MSE (i.e., a negative value), which is why the preceding code computes -scores before calculating the square root.

**cross-validation allows you to get not only an estimate of the performance of your model, but also a measure of how precise this estimate is (i.e., its standard deviation). The Decision Tree has a score of approximately 71,407, generally ±2,439. You would not have this information if you just used one validation set. But cross-validation comes at the cost of training the model several times, so it is not always possible.**

* Random Forests work by training many Decision Trees on random subsets of the features, then averaging out their predictions. Building a model on top of many other models is called **Ensemble Learning**, and it is often a great way to push ML algorithms even further
* try out many other models from various categories of Machine Learning algorithms (e.g., several Support Vector Machines with different kernels, and possibly a neural network), without spending too much time tweaking the hyperparameters. The goal is to shortlist a few (two to five) promising models.

You should save every model you experiment with so that you can come back easily to any model you want. Make sure you save both the hyperparameters and the trained parameters, as well as the cross-validation scores and perhaps the actual predictions as well. This will allow you to easily compare scores across model types, and compare the types of errors they make. You can easily save Scikit-Learn models by using Python’s pickle module or by using the joblib library, which is more efficient at serializing large NumPy arrays (you can install this library using pip):

import joblib

joblib.dump(my\_model, "my\_model.pkl")

# and later...

my\_model\_loaded = joblib.load("my\_model.pkl")

* Randomized search cv and grid search cv are used to find the best combination of hyperparameter values for your model, both are explained below in brief.

## Grid Search

One option would be to fiddle with the hyperparameters manually, until you find a great combination of hyperparameter values. This would be very tedious work, and you may not have time to explore many combinations.

Instead, you should get Scikit-Learn’s GridSearchCV to search for you. All you need to do is tell it which hyperparameters you want it to experiment with and what values to try out, and it will use cross-validation to evaluate all the possible combinations of hyperparameter values. For example, the following code searches for the best combination of hyperparameter values for the RandomForestRegressor: (for detailed explanation on param grid attributes refer book)

from sklearn.model\_selection import GridSearchCV

param\_grid = [

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

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

]

forest\_reg = RandomForestRegressor()

grid\_search = GridSearchCV(forest\_reg, param\_grid, cv=5,#here cv=5 means it will follow 5 fold cross validation

scoring='neg\_mean\_squared\_error',

return\_train\_score=True)

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

###### TIP

When you have no idea what value a hyperparameter should have, a simple approach is to try out consecutive powers of 10 (or a smaller number if you want a more fine-grained search, as shown in this example with the n\_estimators hyperparameter). (for detailed explanation refer to Grid search section in in chapter 2 of Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition for more information)

grid\_search.best\_params\_ #after performing grid search by using this syntax to see best parameters

{'max\_features': 8, 'n\_estimators': 30}

###### TIP

Since 8 and 30 are the maximum values that were evaluated, you should probably try searching again with higher values; the score may continue to improve.

You can also get the best estimator directly:

>>> grid\_search.best\_estimator\_

RandomForestRegressor(bootstrap=True, criterion='mse', max\_depth=None,

max\_features=8, max\_leaf\_nodes=None, min\_impurity\_decrease=0.0,

min\_impurity\_split=None, min\_samples\_leaf=1,

min\_samples\_split=2, min\_weight\_fraction\_leaf=0.0,

n\_estimators=30, n\_jobs=None, oob\_score=False, random\_state=None,

verbose=0, warm\_start=False)

###### NOTE

If GridSearchCV is initialized with refit=True (which is the default), then once it finds the best estimator using cross-validation, it retrains it on the whole training set. This is usually a good idea, since feeding it more data will likely improve its performance.

## Randomized Search

The grid search approach is fine when you are exploring relatively few combinations, like in the previous example, but when the hyperparameter search space is large, it is often preferable to use RandomizedSearchCV instead. This class can be used in much the same way as the GridSearchCV class, but instead of trying out all possible combinations, it evaluates a given number of random combinations by selecting a random value for each hyperparameter at every iteration. This approach has two main benefits:

* If you let the randomized search run for, say, 1,000 iterations, this approach will explore 1,000 different values for each hyperparameter (instead of just a few values per hyperparameter with the grid search approach).
* Simply by setting the number of iterations, you have more control over the computing budget you want to allocate to hyperparameter search.
* Ensemble Methods

Another way to fine-tune your system is to try to combine the models that perform best. The group (or “ensemble”) will often perform better than the best individual model (just like Random Forests perform better than the individual Decision Trees they rely on), especially if the individual models make very different types of errors.

* Analyze the Best Models and Their Errors

You will often gain good insights on the problem by inspecting the best models. For example, the RandomForestRegressor can indicate the relative importance of each attribute for making accurate predictions:

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

>>> feature\_importances

array([7.33442355e-02, 6.29090705e-02, 4.11437985e-02, 1.46726854e-02,

1.41064835e-02, 1.48742809e-02, 1.42575993e-02, 3.66158981e-01,

5.64191792e-02, 1.08792957e-01, 5.33510773e-02, 1.03114883e-02,

1.64780994e-01, 6.02803867e-05, 1.96041560e-03, 2.85647464e-03])

Let’s display these importance scores next to their corresponding attribute names:

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

>>> cat\_encoder = full\_pipeline.named\_transformers\_["cat"]

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

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

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

[(0.3661589806181342, 'median\_income'),

(0.1647809935615905, 'INLAND'),

(0.10879295677551573, 'pop\_per\_hhold'),

(0.07334423551601242, 'longitude'),

(0.0629090704826203, 'latitude'),

(0.05641917918195401, 'rooms\_per\_hhold'),

(0.05335107734767581, 'bedrooms\_per\_room'),

(0.041143798478729635, 'housing\_median\_age'),

(0.014874280890402767, 'population'),

(0.014672685420543237, 'total\_rooms'),

(0.014257599323407807, 'households'),

(0.014106483453584102, 'total\_bedrooms'),

(0.010311488326303787, '<1H OCEAN'),

(0.002856474637320158, 'NEAR OCEAN'),

(0.00196041559947807, 'NEAR BAY'),

(6.028038672736599e-05, 'ISLAND')]

With this information, you may want to try dropping some of the less useful features (e.g., apparently only one ocean\_proximity category is really useful, so you could try dropping the others).

You should also look at the specific errors that your system makes, then try to understand why it makes them and what could fix the problem (adding extra features or getting rid of uninformative ones, cleaning up outliers, etc.).

* Evaluate Your System on the Test Set

After tweaking your models for a while, you eventually have a system that performs sufficiently well. Now is the time to evaluate the final model on the test set. There is nothing special about this process; just get the predictors and the labels from your test set, run your full\_pipeline to transform the data (call transform(), not fit\_transform()—you do not want to fit the test set!), and evaluate the final model on the 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\_predictions = final\_model.predict(X\_test\_prepared)

final\_mse = mean\_squared\_error(y\_test, final\_predictions)

final\_rmse = np.sqrt(final\_mse) # => evaluates to 47,730.2

In some cases, such a point estimate of the generalization error will not be quite enough to convince you to launch: what if it is just 0.1% better than the model currently in production? You might want to have an idea of how precise this estimate is. For this, you can compute a 95% confidence interval for the generalization error using scipy.stats.t.interval():

>>> from scipy import stats

>>> confidence = 0.95

>>> squared\_errors = (final\_predictions - y\_test) \*\* 2

>>> np.sqrt(stats.t.interval(confidence, len(squared\_errors) - 1,

... loc=squared\_errors.mean(),

... scale=stats.sem(squared\_errors)))

...

array([45685.10470776, 49691.25001878])

If you did a lot of hyperparameter tuning, the performance will usually be slightly worse than what you measured using cross-validation (because your system ends up fine-tuned to perform well on the validation data and will likely not perform as well on unknown datasets). It is not the case in this example, but when this happens you must resist the temptation to tweak the hyperparameters to make the numbers look good on the test set; the improvements would be unlikely to generalize to new data.

* most machine learning applications require not only the application of a single algorithm, but the chaining together of many different processing steps and machine learning models.
* the splitting (in to train and test) of the dataset during cross-validation should be done before doing any preprocessing. Any process that extracts knowledge from the dataset should only ever be learned from the training portion of the dataset, and therefore be contained inside the cross-validation loop
* Pipeline class (in scikit learn) itself has fit, predict, and score methods and behaves just like any other model in scikit-learn. The most common use case of the Pipeline class is in chaining preprocessing steps (like scaling of the data) together with a supervised model like a classifier.
* Let’s look at how we can use the Pipeline class to express the workflow for training an SVM after scaling the data with MinMaxScaler (for now without the grid search). First, we build a pipeline object by providing it with a list of steps. Each step is a tuple containing a name (any string of your choosing[**1**](https://learning.oreilly.com/library/view/introduction-to-machine/9781449369880/ch06.html#idm140338225637296)) and an instance of an estimator:

**In[5]:**

from sklearn.pipeline import Pipeline

pipe = Pipeline([("scaler", MinMaxScaler()), ("svm", SVC())])

* pipe.fit(X\_train, y\_train)

Here, pipe.fit first calls fit on the first step (the scaler), then transforms the training data using the scaler, and finally fits the SVM with the scaled data. To evaluate on the test data, we simply call pipe.score:

**In[7]:**

print("Test score: {:.2f}".format(pipe.score(X\_test, y\_test)))

**Out[7]:**

Test score: 0.95

Calling the score method on the pipeline first transforms the test data using the scaler, and then calls the score method on the SVM using the scaled test data. As you can see, the result is identical to the one we got from the code at the beginning of the chapter, when doing the transformations by hand. **Using the pipeline, we reduced the code needed for our “preprocessing + classification” process**. The main benefit of using the pipeline, however, is that we can now use this single estimator in cross\_val\_score or GridSearchCV.

* features can only be selected using the training folds of the data, not the test fold. The feature selection finds features that are correlated with the target on the training set, but because the data is entirely random, these features are not correlated with the target on the test set. In this example, rectifying the data leakage issue in the feature selection makes the difference between concluding that a model works very well and concluding that a model works not at all.
* The Pipeline class is not restricted to preprocessing and classification, but can in fact join any number of estimators together. For example, you could build a pipeline containing feature extraction, feature selection, scaling, and classification, for a total of four steps. Similarly, the last step could be regression or clustering instead of classification.

The only requirement for estimators in a pipeline is that all but the last step need to have a transform method, so they can produce a new representation of the data that can be used in the next step.

Internally, during the call to Pipeline.fit, the pipeline calls fit and then transform on each step in turn,[**2**](https://learning.oreilly.com/library/view/introduction-to-machine/9781449369880/ch06.html#idm140338225176288) with the input given by the output of the transform method of the previous step. For the last step in the pipeline, just fit is called. When predicting(using cross validation or while testing) using Pipeline, we similarly transform the data using all but the last step, and then call predict on the last step

Diagram

Description automatically generated

* Principal Component Analysis is **an unsupervised learning algorithm that is used for the dimensionality reduction in machine learning**.
* **Convenient Pipeline Creation with make\_pipeline:**

Creating a pipeline using the syntax described earlier is sometimes a bit cumbersome, and we often don’t need user-specified names for each step. There is a convenience function, make\_pipeline, that will create a pipeline for us and automatically name each step based on its class. The syntax for make\_pipeline is as follows:

**In[17]:**

from sklearn.pipeline import make\_pipeline

# standard syntax

pipe\_long = Pipeline([("scaler", MinMaxScaler()), ("svm", SVC(C=100))])

# abbreviated syntax

pipe\_short = make\_pipeline(MinMaxScaler(), SVC(C=100))

The pipeline objects pipe\_long and pipe\_short do exactly the same thing, but pipe\_short has steps that were automatically named. We can see the names of the steps by looking at the steps attribute:

**In[18]:**

print("Pipeline steps:\n{}".format(pipe\_short.steps))

**Out[18]:**

Pipeline steps:

[('minmaxscaler', MinMaxScaler(copy=True, feature\_range=(0, 1))),

('svc', SVC(C=100, cache\_size=200, class\_weight=None, coef0=0.0,

decision\_function\_shape='ovr', degree=3, gamma='auto',

kernel='rbf', max\_iter=-1, probability=False,

random\_state=None, shrinking=True, tol=0.001,

verbose=False))]

* **Accessing Attributes in a Pipeline inside GridSearchCV:**

As we discussed earlier in this chapter, one of the main reasons to use pipelines is for doing grid searches. A common task is to access some of the steps of a pipeline inside a grid search. Let’s grid search a LogisticRegression classifier on the cancer dataset, using Pipeline and StandardScaler to scale the data before passing it to the LogisticRegression classifier. First we create a pipeline using the make\_pipeline function:

**In[21]:**

from sklearn.linear\_model import LogisticRegression

pipe = make\_pipeline(StandardScaler(), LogisticRegression())

Next, we create a parameter grid. As explained in [Chapter 2](https://learning.oreilly.com/library/view/introduction-to-machine/9781449369880/ch02.html#supervised-learning), the regularization parameter to tune for LogisticRegression is the parameter C. We use a logarithmic grid for this parameter, searching between 0.01 and 100. Because we used the make\_pipeline function, the name of the LogisticRegression step in the pipeline is the lowercased class name, logisticregression. To tune the parameter C, we therefore have to specify a parameter grid for logisticregression\_\_C:

**In[22]:**

param\_grid = {'logisticregression\_\_C': [0.01, 0.1, 1, 10, 100]}

As usual, we split the cancer dataset into training and test sets, and fit a grid search:

**In[23]:**

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

cancer.data, cancer.target, random\_state=4)

grid = GridSearchCV(pipe, param\_grid, cv=5)

grid.fit(X\_train, y\_train)

So how do we access the coefficients of the best LogisticRegression model that was found by GridSearchCV? From [Chapter 5](https://learning.oreilly.com/library/view/introduction-to-machine/9781449369880/ch05.html#model-evaluation-and-improvement) we know that the best model found by GridSearchCV, trained on all the training data, is stored in grid.best\_estimator\_:

**In[24]:**

print("Best estimator:\n{}".format(grid.best\_estimator\_))

**Out[24]:**

Best estimator:

Pipeline(memory=None, steps=[

('standardscaler', StandardScaler(copy=True, with\_mean=True, with\_std=True)),

('logisticregression', LogisticRegression(C=0.1, class\_weight=None,

dual=False, fit\_intercept=True, intercept\_scaling=1, max\_iter=100,

multi\_class='warn', n\_jobs=None, penalty='l2', random\_state=None,

solver='warn', tol=0.0001, verbose=0, warm\_start=False))])

This best\_estimator\_ in our case is a pipeline with two steps, standardscaler and logisticregression. To access the logisticregression step, we can use the named\_steps attribute of the pipeline, as explained earlier:

**In[25]:**

print("Logistic regression step:\n{}".format(

grid.best\_estimator\_.named\_steps["logisticregression"]))

**Out[25]:**

Logistic regression step:

LogisticRegression(C=0.1, class\_weight=None, dual=False, fit\_intercept=True,

intercept\_scaling=1, max\_iter=100, multi\_class='warn',

n\_jobs=None, penalty='l2', random\_state=None, solver='warn',

tol=0.0001, verbose=0, warm\_start=False)

Now that we have the trained LogisticRegression instance, we can access the coefficients (weights) associated with each input feature:

**In[26]:**

print("Logistic regression coefficients:\n{}".format(

grid.best\_estimator\_.named\_steps["logisticregression"].coef\_))

**Out[26]:**

Logistic regression coefficients:

[[-0.389 -0.375 -0.376 -0.396 -0.115 0.017 -0.355 -0.39 -0.058 0.209

-0.495 -0.004 -0.371 -0.383 -0.045 0.198 0.004 -0.049 0.21 0.224

-0.547 -0.525 -0.499 -0.515 -0.393 -0.123 -0.388 -0.417 -0.325 -0.139]]

This might be a somewhat lengthy expression, but often it comes in handy in understanding your models.

* GridSearchCV tries all possible combinations of the specified parameters. Therefore, adding more parameters to your grid exponentially increases the number of models that need to be built.
* Any transformer we design, inheriting those transformer from BaseEstimator and TransformerMixin classes as they give us pre-existing methods for free. There are 4 methods to take care of here:

1. \_\_init\_\_ - This is the constructor. It is called when pipeline is initialized.
2. fit() - Called when we fit the pipeline.
3. transform() - Called when we use transform on the pipeline.
4. fit\_transform() - Called when we use fit and transform on the pipeline

* if there is a categorical variable(for example sex- will have male, female and other), then sex is called categorical variable and it has 3 levels(male, female and others)
* when you know that a variable is going to contribute majorly in predict outcome in the ml model, you need to use to pd.cut and then make it in to bins and generate a column, and then stratify sampling based on that column:

housing["income\_cat"]= pd.cut(housing["median\_income"],  
bins=[0., 1.5, 3.0, 4.5, 6., np.inf],  
labels= [1,2,3,4,5])  
housing["income\_cat"].hist()

# Startified sampling based on income\_cat to make the datasets more random and representative  
  
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(housing, housing["income\_cat"]): (**NOTICE THE INCOME\_CAT COLUMN BEING USED HERE TO PERFORM STRATIFIED SAMPLING**)  
strat\_train\_set = housing.loc[train\_index]  
strat\_test\_set = housing.loc[test\_index]

* there is more to 2d plot than just viewing data in 2 dimensions, let’s say you are plotting **population size(the bigger the population, the bigger the size)** in circles using longitude and latitude along x and y axis, and in side you want to have the **price bar** which will have **the colour (the higher the price of the house, the colour of the circle is changed accordingly)**

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()

s=decides size of circle

c=decides color of circle

Chart, scatter chart

Description automatically generated

* correlation with target variable shows how important might the variable be:

*## How other variables relate with our target variable*  
  
corr\_matrix = housing.corr()  
corr\_matrix["median\_house\_value"].sort\_values(ascending=False)

Table

Description automatically generated

Correlation Table

Our usual suspects, median income, total rooms and age are top 3 variables in terms of correlation with our target variable. We can even look at the correlation plots.

from pandas.plotting import scatter\_matrixattributes = ["median\_house\_value", "median\_income", "total\_rooms",  
 "housing\_median\_age"]  
scatter\_matrix(housing[attributes], figsize=(12, 8))  
save\_fig("scatter\_matrix\_plot")

Graphical user interface, diagram

Description automatically generated with medium confidence

Correlation plots

* impute.fit is done on train data and impute.transform is done on both train and test data
* data leakage happens when we have some instances of data which is there in train data is similarly the same data in validation or test data
* from pandas.plotting import scatter\_matrix

this scatter matrix will do scatter plots on list of columns that are passed if you have 4 columns, you will have 16 scatter plots as shown above

* encoding/transforming for categorical values is not needed for tree based algorithms(decision tree,rfc), in else cases(linear regression and logistic regression)
* imputing can be done in many types, it can be mean, median and many other, if you think other features contribute of the current row contribute to missing value, then KNN imputation might be a good approach
* you can check mean, median and standard deviation on an numpy array which is a result of many data transformation results such as cross validation scores.. and so on
* encoding is basically of 2 types
* ordinal(ordinal stands for order):

in which the order is to be preserved, i.e, ex1: best>great>good, ex2: A+>A>B+>B

* Nominal(Nominal stands for name):

In which we have to encode names, ie, ex: continents in earth- asia,america, Australia..

One hot encoding is Nominal

* Column transformer can be used when you have a dataframe in which we need to apply different transformations on different columns then you can use column transformer.

from sklearn.pipeline import Pipeline

from sklearn.preprocessing import StandardScaler

num\_pipeline = Pipeline([

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

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

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

])

housing\_num\_tr = num\_pipeline.fit\_transform(housing\_num)

from sklearn.compose import ColumnTransformer

num\_attribs = list(housing\_num)

cat\_attribs = ["ocean\_proximity"]

full\_pipeline = ColumnTransformer([

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

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

])

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

here in the above example, num\_pipeline is a **custom pipeline of pipelines** where the pipeline logic is applied to all columns in dataframe. Then we applied column transformer num is applied on dataframe where num\_pipeline is applied on numerical attributes, and one hot encoder on categorical attributes, the fit\_transform is applied on the total dataframe.

