My Notes to Aurélien Géron's *Hands-on Machine Learning with Scikit-Learn, Keras, & TensorFlow*

Eric Xia

Last Updated 17 September 2020

Contents

1 The Machine Learning Landscape

Machine Learning:
• The field of study that gives computers the ability to learn without being explicitly programmed
• "A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E" — Tom Mitchell

Example:
If we were to create a ML spam filter by giving the computer examples of spam and non-spam emails, referred to as **training instances (samples)**. In this scenario, the task T is flagging spam/not- spam in new emails, the experience E is the training data, and the performance measure P needs to be defined, for example, P could be the ratio of correctly classified emails. P is also referred to as **accuracy** and is often used in classification-type problems.

The advantages of implementing machine learning in a project may include the ability for the project to generalize to scenarios it has never seen before, being able to solve problems that are too complex for traditional approaches or have no known algorithm, and help humans learn. Sometimes we may apply ML techniques to analyze paterns in a large dataset that were not immediately apparent. This is called **data mining**.

Types of Machine Learning Systems:
• Whether or not they are trained under human supervision (**supervised, unsupervised, semisupervised,** and **reinforcement learning**)
• Whether they can process learn in real-time (**online** vs. **batch learning**)
• Whether they work by comparing new data points to known data points or instead by detecting patterns in the training data and building a predictive model (**instance-based** vs. **model-based learning**)

In **Supervised Learning**, we feed the algorithm a training set with the desired solutions, called labels. A common supervised learning task is **classification**, where the algorithm is trained with many example samples along with their class and thus the algorithm must learn how to classify new examples. Another common supervised leanring task is to predict a **target** numeric value, such as the price of a car, given a set of **features** (mileage, age, brand, etc.) called **predictors**. This type of task is referred to as **regression**. In ML, **attributes** and **features** may be used interchangingly, however, *attributes* are often a data type, say, mileage, and *features* are an attribute with a corresponding value. Note that some regression algorithms can be used for classification problems, and vice versa.

Some of the Most Important Supervised Leanring Algorithms:
• $k-$Nearest Neighbors
• Linear Regression
• Logistic Regression
• Support Vector Machines (SVMs)
• Decision Trees and Random Forests
• Neural Networks

In **Unsupervised Learning**, the training data used is unlabeled and the system tries to learn on its own.

Some of the Most Important Unsupervised Learning Algorithms:
• Clustering
- K-Means
- DBSCAN
- Hierarchical Cluster Analysis (HCA)
• Anomaly Detection and Novelty Detection
- One-class SVM
- Isolation Forest
• Visualization and Dimensionality Reduction
- Principal Component Analysis (PCA)
- Kernel PCA
- Locally Linear Embedding (LLE)
- t-Distributed Stochastic Neighbor Embedding (t-SNE)
• Association Rule Learning
- Apriori
- Eclat

Clustering algorithms attempt to detect groups of similar visitors. **Hierarchical clustering** algorithms may subdivide each group into smaller groups. **Dimensionality Reduction** attempts to simplify a dataset without losing significant information, often by merging multiple correlative features into one. **Anomaly Detection** is often used to detect fraud through unusual credit card transactions. Similarly, **Novelty Detection** attempts to detect new instances different from instances in the training set. In **Association Rule Learning**, we try to analyze large datasets for interesting relations between attributes.

In **Semisupervised Learning**, we feed algorithms partially-labelled data.

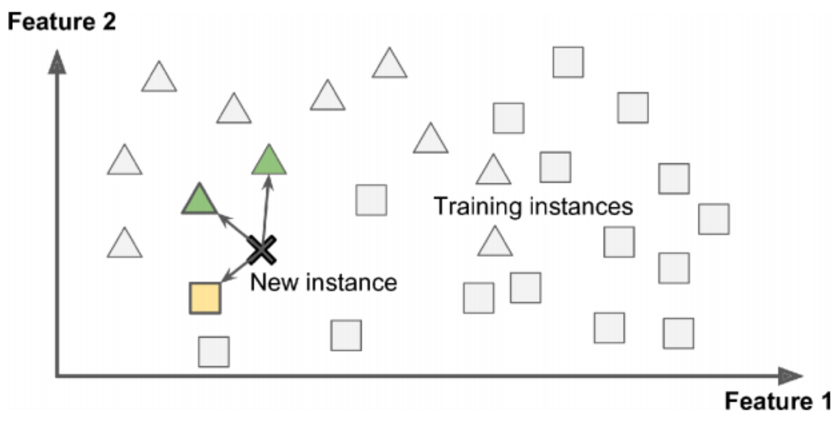
In **Reinforcement Learning**, the learning system, called an **agent**, can observe the environment, select and perform actions, and receive **rewards/penalties** in return. It then learns by itself what the best strategy, called a **policy**, is, to receive the greatest long-term reward.

In **Batch Learning**, the system cannot learn incrementally and it must be trained using all of the available data. This usually takes a lot of time and computing resources so it is usually performed offline. The system is first trained, then launched into production and runs without further learning. This is also referred to as **offline learning**. Batch Learning can be very annoying when you want to implement new data as you have to retrain the entire system from scratch each time.

In **Online Learning**, the system is trained incrementally by feeding it data instances sequentially, either individually or in small groups called **mini-batches**. Online Learning is particularly useful for systems that receive a continuous flow of data and need to adapt to change rapidly or autonomously. Online learning algorithms can be used to train systems with huge datasets larger than the computer's main memory. We refer to this as **out-of-core learning**, and it is important to note that out-of-core learning is usually performed offline. Online learning systems have an influencing variable called the **learning rate**, which determines how fast the system should adapt to the changing data. Systems with high learning rates wil be able to adapt quickly, but they will also forget old data faster. Conversely, if we set a low learning rate, then the system will learn slowly but will be less sensitive to noise in the new data or to outliers. It is important to note that if bad data is fed to a system that has implemented online learning, then the system's performance will wane and clients may notice this. Hence, it is suggested to monitor the input data and react to abnormal data through an anomaly detection algorithm.

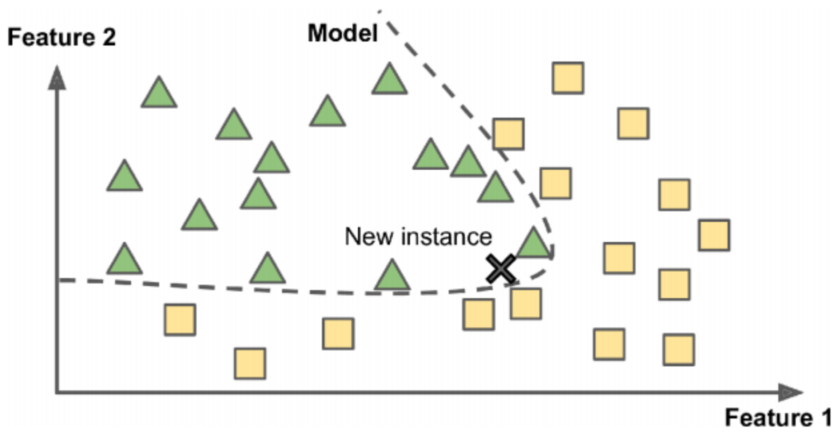
In Machine Learning, we want to be able to generalize both training data measurements and predictions of data. Two main approaches to this are instance-based learning and model-based learning:

Instance-based Learning:
The system learns the examples by heart, then generalizes to new cases by using a **similarity measure** to compare them to learned examples. In the context of a spam filter, our system would flag all emails that are identical to emails that have been previously flagged by users. We could also set up the system so that it flags emails that are very similar to known spam email, a process that requires a measure of similarity. In the figure below, the new instance would be classified as a triangle since the majority of the most similar instances below to that class.



Instance-based Learning

Model-based Learning:
A model of existing examples is made and applied to make predictions.

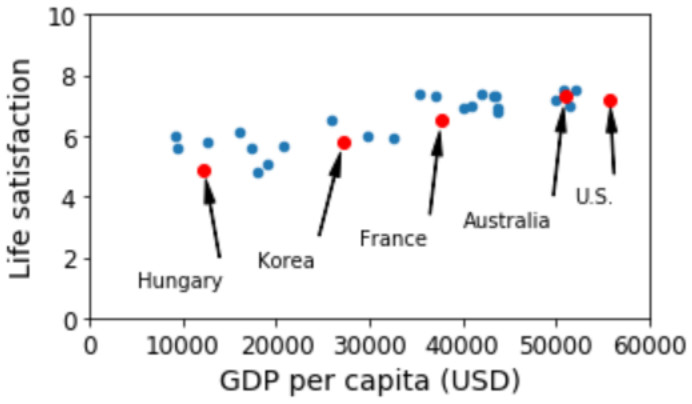


Model-based Learning

Let's say we wanted to know if money makes people happier, so we downloaded the Better Life Index data from the OECD's website and stats about GDP per capita from the IMF's website. Joining the tables, we get:

Country	**GDP/Capita in USD**	**Life Satisfaction**
Hungary	12,240	4.9
Korea	27,195	5.8
France	37,675	6.5
Australia	50,962	7.3
United States	55,805	7.2
...	...	...

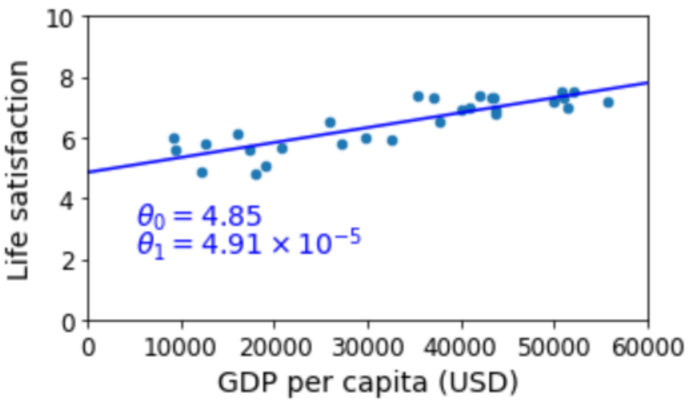
Plotting this data, we get



Although the visualized data is somewhat noisy, it does look like life satisfaction and GDP/Capita share a positive correlational relationship. We will choose to model life satisfaction as a linear function of GDP/Capita. This step is called **model selection** and we have selected a **linear model** of life satisfaction with one attribute, GDP/Capita:

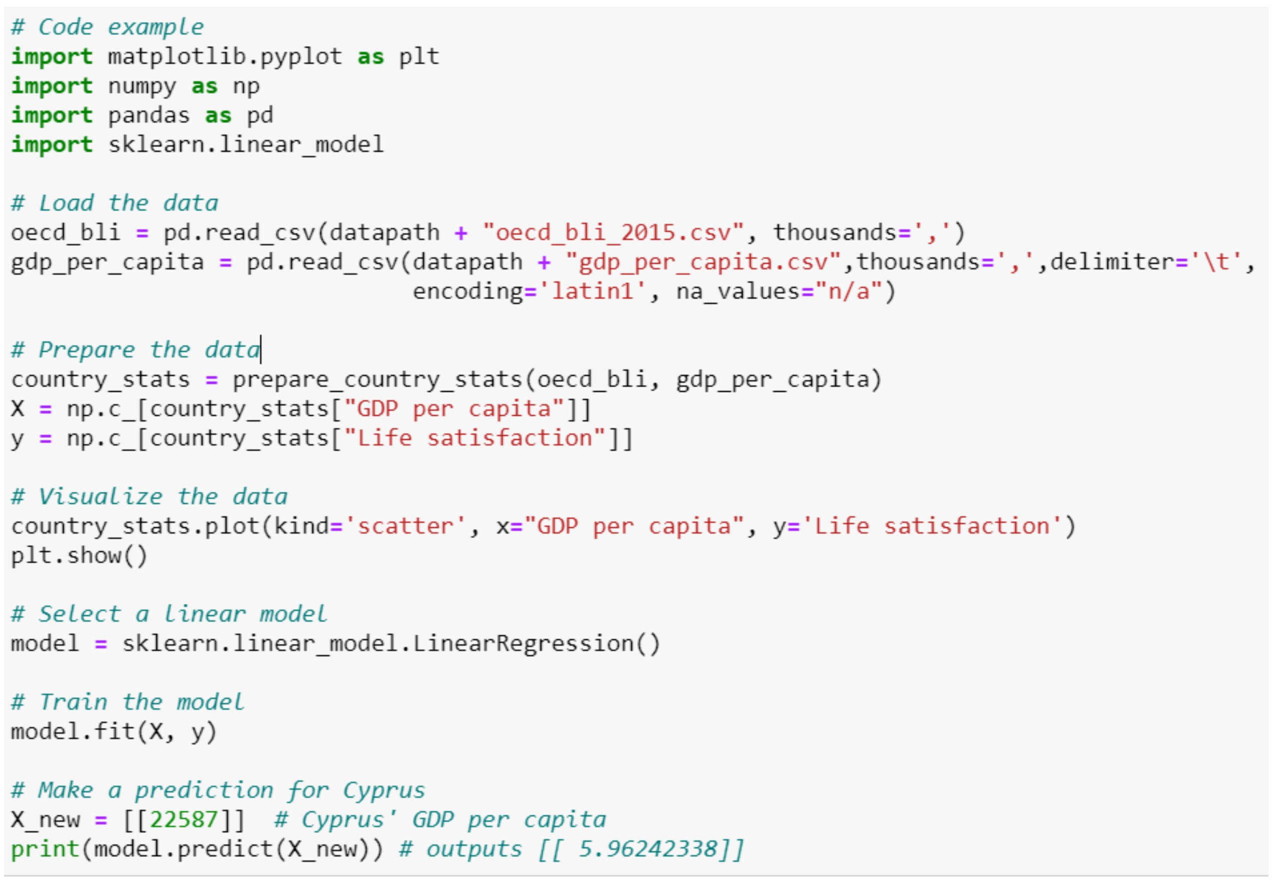
$$\text{life\_satisfaction } = \theta_0 + \theta_1 \times \text{ GDP\_per\_capita}$$

This model has two **model parameters**, θ_0 and θ_1. By changing the values of these parameters, we can make our model represent any linear function. Before we can use our model, we should try to define values of θ_0 and θ_1 that will give us the optimum performance. This performance is either measured by an **utility function**, which measures how *good* a model is, or a **cost function**, which measures how *bad* a model is. For Linear Regression problems, it is conventional to use a cost function that measures the distance between the linear model's predictions and the values of the actual training examples; the objective is to minimize this distance. In this example, the algorithm finds the optimal parameter values to be $\theta_0 = 4.85$ and $\theta_1 = 4.91 \times 10^{-5}$. The figure below displays the linear model and the training examples.



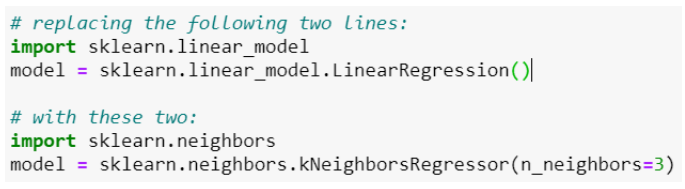
Now we can finally run the model to make some predictions. If we ever wanted to know how happy Cypriots but the OECD data doesn't have Cyprus's data, we can plug Cyprus's GDP/Capita, \$22,587 into the model. We will then find the life satisfaction to be somewhere around $4.85 + 22,587 \times 4.91 \times 10^{-5} = 5.96$.

Below is the python code necessary to load the data, prepare it, create a scatterplot for visualization, and train a linear model and make a prediction:



```
# Code example
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import sklearn.linear_model

# Load the data
oecd_bli = pd.read_csv(datapath + "oecd_bli_2015.csv", thousands=',')
gdp_per_capita = pd.read_csv(datapath + "gdp_per_capita.csv",thousands=',',delimiter='\t',
                             encoding='latin1', na_values="n/a")

# Prepare the data
country_stats = prepare_country_stats(oecd_bli, gdp_per_capita)
X = np.c_[country_stats["GDP per capita"]]
y = np.c_[country_stats["Life satisfaction"]]

# Visualize the data
country_stats.plot(kind='scatter', x="GDP per capita", y='Life satisfaction')
plt.show()

# Select a linear model
model = sklearn.linear_model.LinearRegression()

# Train the model
model.fit(X, y)

# Make a prediction for Cyprus
X_new = [[22587]]  # Cyprus' GDP per capita
print(model.predict(X_new)) # outputs [[ 5.96242338]]
```

If we had instead used an instance-based learning algorithm, we would have found that Slovenia has the closest GDP/Capita to that of Cyprus ($20,732), and since hte OECD data tells us that Slovenians' life satisfaction is 5.7, we would have predicted a life satisfaction of 5.7 for Cyprus. If we zoom out slightly and look at the two next-closest countries, we will find Portugal and Spain with life satisfactions of 5.1 and 6.5, respectively. We average these three values to get 5.77, which is fairly close to the prediction made by our model. This algorithm is called the **k-Nearest Neighbors** regression. We can replace the linear regression model with k-Nearest Neighbors regression in the previous code by:



```
# replacing the following two lines:
import sklearn.linear_model
model = sklearn.linear_model.LinearRegression()

# with these two:
import sklearn.neighbors
model = sklearn.neighbors.KNeighborsRegressor(n_neighbors=3)
```

The Main Challenges of Machine Learning: insufficient quantity of training data, nonrepresentative training data, poor-quality data, irrelevant features, and overfitting/underfitting the training data

Insufficient Quantity of Training Data:
Even for very simple problems, you typically need thousands of examples, and for complex problems such as image or speech recognition you may need millions of examples, unless you can reuse components of existing models. A famous paper published by MSR in 2001 showed that, surprisingly, ML algorithms, including simple ones, performed almost identically on a complex problem of natural language disambiguation. The results of the study suggest that data should be considered more important than the particular algorithm implemented.

Nonrepresentative Training Data:
No matter if you use instance-based learning or model-based learning, it is important that your training data be representative of new cases you want to generalize to. Below is a plot of the training data in our earlier example, our model linear function, and predicted samples for seven new nations. If you train a linear model on this data, you get the solid line, whereas the old model that did not have additional countries is represented by the dotted line. It is fairly obvious that adding a few missing countries significantly changes the model and a simple linear model is likely to never work well, as the model incorrectly suggests that very rich countries are not happier than moderately rich countries and very poor countries seem happier than many rich countries. Hence, this nonrepresentative training set was unlikely to generalize accurately. If the sample is too small then you will have **sampling noise**, which is nonrepresentative data as a result of chance, but even very large samples can be nonrepresentative if the sampling method is flawed. This is referred to as **sampling bias**.

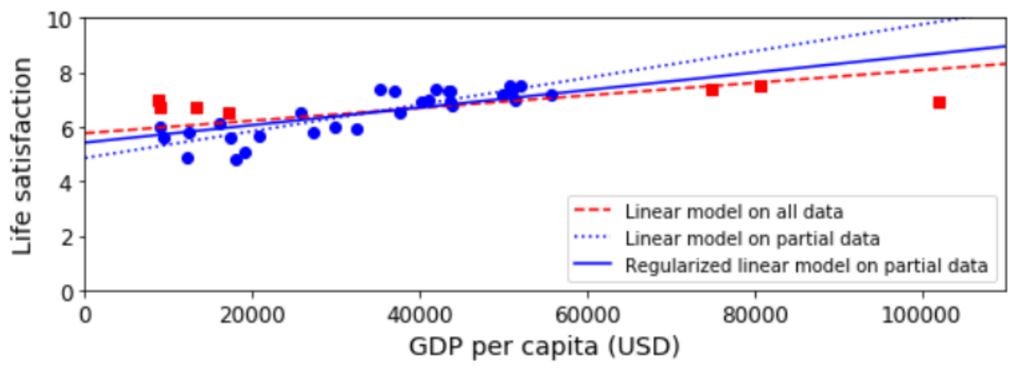
Poor Quality Data:
If some instances are clearly outliers it may help to get rid of them or try to manually fix the errors. If some instances are missing a few features (i.e. some respondents chose to not specify their age), then you should decide if you wish to ignore this attribute, ignore these instances, fill in the missing values with the median age, or train a model with the feature and one model without it.

Irrelevant Features:
The process of deciding a good set of features to train the model on is called **feature engineering**, which involves the following steps:
1. **Feature selection:** choosing the most useful features to train on among existing features
2. **Feature extraction:** combining existing features to produce a more useful one
3. Creating new features by gathering new data

Overfitting the Training Data:
Overfitting is when the model performs well on the training data but does not generalize well. For example, if you were visiting a foreign country and a taxi driver rips you off it would be an overgeneralization (result of overfitting) to say that all the taxi drivers in that country are thieves. Deep neural networks and other complex models can detect subtle patterns in the data but if the training set is noisy or too small (introduces sampling noise) then the model is likely to detect patterns in the noise itself.

Possible Solutions For Overfitting:
• Simplify the model by selecting one with fewer parameters, by reducing the number of attributes, or by constraining the model.
• Make the training set larger
• Reduce the noise in the training data (e.g., fix data errors and remove outliers)

Regularization is the constraining of a model to make it simpler and reduce the risk of overfitting. Earlier, we used a linear model with two parameters or two **degrees of freedom**, θ_0 and θ_1. These degrees of freedom can adjust both the height (θ_0) and the slope (θ_1) of the line. Through regularization, if we force $\theta_1 = 0$ then the algorithm will only have one degree of freedom and would have a much harder time fitting the data properly; it could only move the line up and down to approach the training instances, so it would end up around the mean. If we let the algorithm to modify θ_1 but we force it to keep it small, then the learning algorithm will effectively have somewhere in between one and two degrees of freedom, hence producing a model that's simpler than one wth two degrees of freedom, but more complex than one with just one. When using regularization, try to find a balance between fitting the training data perfectly and keeping the model simple enough to ensure that it will generalize well. Below is a figure that shows all three models. Note how regularization forced the model to have a smaller slope, making it perform better in generalization.



Regularization Reduces the Risk of Overfitting

We can control the amount of regularization to apply by a **hyperparameter**, which is a parameter of a learning algorithm distinct from the model. Hence, it has to be set before training and remains constant during training. If you set the hyperparameter to a high value you will get a nearly flat model. Correctly choosing hyperparameters will be covered in the next chapter.

Underfitting the Training Data:
Underfitting occurs when the model used is too simple to learn the underlying structure of the data.

Possible Solutions for Underfitting:
• Choose a more powerful model with more parameters
• Feed better features to the learning algorithm (feature engineering)
• Reduce the constraints on the model (e.g., reduce the regularization hyperparameter)

Testing and Validating:
To understand how well a model will generalize to new cases, we can split our data into two sets: the **training set** and the **test set**. Conventionally, 80% of the data is used for training and %20 is kept for testing. However, this depends on the size of the dataset; if you have an extremely large dataset then smaller percentages may be sufficient for testing. The error rate on new cases is called the **generalization error**, or the **out-of-sample error**, and by evaluating your model on the test set you can get an estimate of the error. *If the training error is low but the generalization error is high then your model is overfitting the training data.*

Hyperparameter Tuning and Model Selection:
A common solution to the problem of having to use different hyperparameters for each unique set is **holdout validation**, in which you hold out part of the training set to evaluate multiple candidate models and choose the best one. The new held-out set is called the **validation/development/dev set**. You train multiple models with different hyperparameters on the full training set minus the validation set and choose the model that performs best on hte validation set. Another way to solve this would be to use **cross-validation**, which takes

many small validation sets, evaluating models for each set after training. Then the evaluations of the model are averaged and the picture of performance is more accurate.