Data Science + Machine Learning

isaiah baksh

June 26 2025

Introduction

I am making this as a reference and as learning notes for anyone trying to learn where and how to start in Machine Learning and Data Science, I've always been interested in it, but just do not know where or how to start. I am hoping to learn more as I go about writing this, and get better at typesetting and improving my programming in python in every way possible.

Most of the material I write in here is going to be a composition of knowledge from the books Hands on Machine Learning, By Aurelion Geron, and Dive into Data Science by Bradford Tuckfield, I am either taking ideas from these textbooks and expanding on them, and learning more about them, or some of the content written, is even me answering the exercises in the books.

Eventually after Data Science + Machine Learning techniques my goal is to add more on Time Series Analysis and maybe even make one on Stochastic Differential Equations, maybe I even use this document one day for a course I teach at a university who knows but I should not have doubt, this is going to be useful for myself and anyone else entering the workforce in a role involving data science or machine learning.

1 Machine Learning Landscape (Geron)

Supervised Learning: A training set is fed to an algorithm that includes its desired solutions. Typically, classification or regression falls under supervised learning. More specifically, Linear and Logistic Regression, SVM's, Decision Trees, and Random Forests.

Unsupervised Learning: Training data is unlabeled, and the algorithm tries to classify itself. Some algorithms include K-means clustering, spectral clustering, and anomaly detection algorithms such as isolation forests and one-class SVM's.

Reinforcement Learning: The algorithm begins to learn its environment on its own.

Batch Learning: The system cannot learn incrementally, and is trained on all available data, the system is trained and launched, and then runs without learning anymore (Also called offline learning).

Instance Based Learning: Systems learn from the data and then it generalizes it to new data using something called a similarity measure.

Model Based Learning: Building a model from previous examples and then generalizing to make predictions.

Some other important terms that may come up are Utility Function which measures how good our model performs, and the opposite to that is a Cost Function which measures how bad our model performs.

Exercises

Answers to the exercises in the Geron textbook

1. Machine Learning can be thought of as writing a program that allows a computer to learn from the data it is given, and then that data is able to produce a model that can do a wide variety of tasks like predicting a value, or classify something into a certain category.
2. Machine Learning shines in area such as, fraud detection, forecasting revenue and sales, recommending certain products for people in advertising.
3. Labeled data provides description to the data point we would be looking at, important for providing context for the data that we are analyzing. While Unlabeled data usually does not come with any tags at all or important characteristics or context.
4. The two common tasks in supervised machine learning is classification, or regression.
5. Four common tasks in unsupervised machine learning is, clustering, anomaly detection, novelty detection and dimensionality reduction. (Note: These were mentioned in gerons book as were the ones above)
6. Reinforcement learning would be ideal, to train a robot to adapt to its environment, as you could penalize it or reward it for depending on its interactions int he environment it is in.
7. Clustering techniques would be ideal to segment customers into certain groups, so we can understand which customers are similar.
8. Spam filtering for emails is usually a classification task, hence we can use supervised learning for this kind of task.
9. Online Learning is when you train the system incrementally or bit by bit and are good for systems that need to adapt rapidly, or systems that receive a continuous flow of data.
10. Out of core learning, is usually done offline similar to online learning, and is used when the data cannot be trained completely on the computers RAM which is why it is done incrementally, it is usually used for extremely large datasets.
11. Instance Based learning relies on similarity measures to make predictions.
12. A Hyperparameter control the learning process, while parameters are variables from the model that are learned from the data during training.

2 End to End Machine Learning Project (Geron)

In this section of the book, a lot of the work is done in Jupyter notebook, I will try to provide some code snippets where I can and if it is applicable, and possibly try to provide the code for the solution to the exercises in here.

It is important to note when attempting to randomly sample, it could result in a test or training set being skewed which would not make the model be quite as accurate because it might be skewed more to one direction. This is why we do something called Stratified Sampling, which would allow us to get a better understanding of the population we are modeling. We are able to implement stratified sampling and random sampling in scikit learn by using from `sklearn.model.selection` we can import `StratifiedShuffleSplit` and `train_test_split` respectively. The population is divided into subgroups known as strata, and then we would select random samples from each strata, to get a better representation of the population for our model.

Scaling

There is two very common ways to implement feature scaling using Scikit-Learn, the first is by using `MinMaxScaler()` and the other is by using just the `StandardScaler()`. Min-Max Scaling transforms all the values to be between 0 and 1, this is also known mathematically as *normalization*. The other method the stand scaler, is called *standardization* in mathematics, and it just centers the data by having a zero mean, and it does this by subtracting each data point by the mean of all of the data points. Both are quite common, but for PCA which we get to below, standardization is used more commonly.

3 Regression Analysis

This section takes from Chapter 4 from the Geron book and some of Chapter 2 from Tuckfield book and even some of my notes from my MATH 4042U class.

It is important to note that their is many different types of regression models, for example, we are first looking at linear regression, and we are using the method of ordinary least squares to fit the regression line to our data points.

Linear Regression (Linear Least Squares)

There are different formulations for Linear Regression through the use of Linear Least Squares, we are specifically below implementing Ordinary Least Squares (OLS), in various different ways. Other methods to implement include Weighted Least Squares and Generalized Least Squares. You might also hear OLS is used in Univariate Linear Regression.

Style	**Equation**	**Used In**
Statistical	$y = \beta_0 + \beta_1 x$	Traditional regression/statistics
ML-style	$y = \theta_0 + \theta_1 x$	Machine learning
Matrix form	$y = \boldsymbol{\theta}^\top \mathbf{x}$	Code and theory in ML

Table 1: Different notations for linear regression models

Above is the three different ways of writing the linear regression model that may come up in different texts. In reality, though we are trying to find the values for θ_0 and θ_1 by using the normal equation, given by equation (1).

$$\hat{\theta} = (\boldsymbol{X}^\top \boldsymbol{X})^{-1} \boldsymbol{X}^\top \boldsymbol{y} \tag{1}$$

In Table 1 it is important to note θ_0 represents the y-intercept and θ_1 represents the slope. The way we find those values of θ is by solving using the normal equation. The point of the Normal Equation is to minimize the error. In statistics, the way to solve for the coefficients in table 1 is by using the formulas below

$$\beta_1 = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sum(x_i - \bar{x})^2} \qquad \textit{and} \qquad \beta_0 = \bar{y} - \beta_1 \bar{x}$$

β_1 is known as the slope equation while β_0 is the y-intercept.

```python
import numpy as np
import matplotlib.pyplot as plt

X = np.random.rand(10,1)
y = 2 + 4 * np.random.rand(10,1)

thetas = np.linalg.inv(X.T @ X) @ X.T @ y
print(thetas) #not entirely accurate

X_bias = np.c_[np.ones((10, 1)), X]
new_thetas = np.linalg.inv(X_bias.T @ X_bias) @ X_bias.T @ y

X_new = np.array([[0], [2]])
X_new_b = np.c_[np.ones((2,1)), X_new]

y_predict = X_new_b @ new_thetas
print(y_predict)

plt.plot(X_new, y_predict, 'r-', label='prediction')
plt.plot(X, y, 'b.')
plt.axis([0,2,0,15])
plt.legend()
plt.show()


from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(X, y)

print(lin_reg.intercept_, lin_reg.coef_)
print(lin_reg.predict(X_new))

theta_best_svd, residuals, rank, s = np.linalg.lstsq(X_bias, y, rcond=1e-6)
print(theta_best_svd)
```

The above implementation includes three different ways of implementing ordinary least squares in python. The first implementation, that is not entirely accurate is implementing the model, where it starts at the origin and omits the y-intercept so the equation will look like y = θ_1x, which is not correct because it does not fit the data correctly. Instead in the next line the `X_bias` term we are adding a column of ones to the original X matrix that contains our information in order to properly compute in the `X_new` our values for θ_0 and θ_1, so the equation for the linear regression model will look like $\hat{y} = \theta_0 \cdot 1 + \theta_1 x$, this results in us having the slope and the y-intercept accurately computed, and in the following lines, we are able to form the regression line between x = 0 and x = 2. Remember in `X_new_b` we have to create a column of 1's first in order to compute the y-intercept like we did above, it is just how matrix multiplication works. The final two methods are quite similar, as we are just using the `sklearn.linear_model` to import their Linear Regression model which just does exactly what I have done before, but hidden, and uses SVD to compute much quicker. the other method using `np.linalg.lstsq` also does the same thing as the sklearn model, using SVD to compute, we are able to also find the rank of the matrix and the singular values of the matrix X aka our data.

Gradient Descent Methods

Just like in OLS our goal is to minimize the cost function, while also trying to fit our regression line to the data points. But instead of solving an equation, it does it iteratively. There are 3 different types of Gradient Descent we are going to look at Batch, Stochastic and Mini Batch. The reason we may choose to use Gradient Descent over a more traditional method of solving is because of time complexity when the data gets significantly larger.

Batch Gradient Descent

The equation for batch gradient descent is broken up into two parts given below

$$\frac{\partial}{\partial \theta_j} MSE(\boldsymbol{\theta}) = \frac{2}{m} \sum_{i=1}^{m} (\boldsymbol{\theta}^\top \boldsymbol{x}^{(i)} - y^{(i)}) x_j^{(i)} \tag{2}$$

$$\boldsymbol{\theta}^{(next)} = \boldsymbol{\theta} - \eta \nabla_\theta MSE(\boldsymbol{\theta}) \tag{3}$$

Equation (2) is the partial derivatives of the cost function, and Equation (3) the gradient descent step. You can also compute all the gradients at once using the equation (4) below.

$$\nabla_\theta \text{MSE}(\theta) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \text{MSE}(\theta) \\ \frac{\partial}{\partial \theta_1} \text{MSE}(\theta) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \text{MSE}(\theta) \end{pmatrix} = \frac{2}{m} \mathbf{X}^\top (\mathbf{X}\theta - \mathbf{y}) \tag{4}$$

We are able to compute all the gradients at once using Equation (4). The whole point of batch gradient descent is that we are selecting a random starting point for θ and we are just iteratively trying to find the most optimal θ's rather than directly attempt to solve the normal equation which takes much more computing power and time to solve as opposed to this. The stopping criterion for Batch Gradient Descent, is either we have a set number of iterations it must go through, or when the norm becomes very small, we are able to set a tolerance usually denoted by ϵ so that we stop solving when it gets that small.

Stochastic Gradient Descent

Gradients are computed at random using only a single instance at a time unlike in batch, making it faster than batch. This is very good for extremely large datasets. With Stochastic Gradient Descent the cost function jumps all over the place as opposed to batch where we see a smooth decrease until we reach the minimum of our cost function, with SGD, the cost function will approach the minimum but never really stop, it will continue to move, making this a good but not an optimal algorithm. In SGD, the learning rate is not a constant like in batch, but is uses a learning schedule, so it formulates a new learning rate at each iteration. The point of this is for the steps in the algorithm to start out quite large and then eventually get much smaller so the algorithm could settle on a minimum. Each round of iterations in SGD is called an *epoch*, which will be seen in our code and is often used with other machine learning algorithms.

Mini Batch Gradient Descent

This variation of gradient descent, is somewhat of a combination of both stochastic and batch gradient descents. It works by computing small random sets of gradients during each iteration which we call mini batches. Just like with stochastic gradient descent it will reach near the minimum, but not compute the exact solution like how we would be able to with the normal equation, or even be as accurate as just batch gradient descent. Mini Batch is good for getting a performance boost though from hardware optimization of matrix operations when using GPU's. Mini Batch is also very good when we have a very large dataset.

```python
#Batch Gradient Descent
import numpy as np

eta = 0.1  # learning rate
n_iterations = 1000
m = 100

theta = np.random.randn(2,1)  # random initialization

for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients

print(theta)
print(X_new_b @ theta)

#Stochastic Gradient Descent
n_epochs = 100
t0, t1 = 5, 50
m = len(X_b)

theta_path_sgd = []

def learning_schedule(t):
    return t0 / (t + t1)

theta_st = np.random.randn(2,1)
```

```
for epoch in range(n_epochs):
    for i in range(m):
        if epoch == 0 and i < 20:
            y_predict = X_b.dot(theta_st)
        random_index = np.random.randint(m)
        xi = X_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(theta_st) - yi)
        eta = learning_schedule(epoch * m + i)
        theta_st = theta_st - eta * gradients
        theta_path_sgd.append(theta_st)

print(theta_st)

from sklearn.linear_model import SGDRegressor

sgd_reg = SGDRegressor(max_iter=100, tol=1e-3, penalty=None, eta0=0.1)
sgd_reg.fit(X, y.ravel())

print(sgd_reg.coef_, sgd_reg.intercept_)

#Mini Batch Gradient Descent
theta_mbgd_path = []

n_iterations = 50
minibatch_size = 20

theta_mbgd = np.random.randn(2,1)

t0, t1 = 200, 1000

def learning_schedule(t):
    return t0 / (t + t1)

t = 0

for epoch in range(n_iterations):
    shuffled_indices = np.random.permutation(m)
    X_b_shuffled = X_b[shuffled_indices]
    y_shuffled = y[shuffled_indices]
    for i in range(0, m, minibatch_size):
        t += 1
        xi = X_b_shuffled[i:i + minibatch_size]
        yi = y_shuffled[i: i + minibatch_size]
        gradients = 2/minibatch_size * xi.T.dot(xi.dot(theta_mbgd) - yi)
        eta = learning_schedule(t)
        theta_mbgd = theta_mbgd - eta * gradients
        theta_mbgd_path.append(theta_mbgd)

print(theta_mbgd)
```

The implementation above uses the same data, from the subsection before.

Polynomial Regression

Not all models are linear, which is why we are able to use a linear model to fit non linear data by adding powers of the features as new features. We can then train our model on this new set of features, that are an extension of the original data and formulate a better more accurate prediction. Using equation (1), but extending the size of the $\boldsymbol{X}$ to incorporate x_i^2 and even higher powers, we are able to do polynomial regression. The form of the model for polynomial regression is given below.

$$y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 \ldots + \beta_m x_i^m$$

Regularized Models

These types of models are also part of linear regression they just add an additional term with the goal in mind to prevent under or over fitting of data points.

Ridge Regression

This regularized model has a term that is added to the cost function, with the goal of keeping the weights of the model as small as possible and is added during training. The hyperparameter given by α is how we control how much we want to regularize the model. The $\alpha = 0$, we just have linear regression, but if it is very large it will result in a flat line with the regression line going through the mean of our data.

$$L(\boldsymbol{\theta}) = MSE(\boldsymbol{\theta}) + \alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2 \tag{5}$$

Equation (5) is our ridge regression cost function, Equation (6) is Ridge Regression with the added term that applied regularization to the model.

$$\hat{\boldsymbol{\theta}} = (\boldsymbol{X}^\top \boldsymbol{X} + \alpha \boldsymbol{A})^{-1} \boldsymbol{X}^\top \boldsymbol{y} \tag{6}$$

It is important to note that the matrix $\boldsymbol{A}$, is a slightly different version fo the identity matrix where the first entry in the diagonal is 0 instead of 1.

Lasso Regression

Similar to Ridge Regression, except that it uses the ℓ_1 norm (Manhattan Distance) of the weight vector as opposed to the ℓ_2 norm used above.

$$L(\boldsymbol{\theta}) = MSE(\boldsymbol{\theta}) + \alpha \sum_{i=1}^{n} |\theta_i| \tag{7}$$

Equation (7) is the cost function for lasso regression.

$$\mathbf{g}(\boldsymbol{\theta}, \lambda) = \nabla_{\boldsymbol{\theta}} \text{MSE}(\boldsymbol{\theta}) + \alpha \begin{pmatrix} \text{sign}(\theta_1) \\ \text{sign}(\theta_2) \\ \vdots \\ \text{sign}(\theta_n) \end{pmatrix}, \quad \text{where } \text{sign}(\theta_i) = \begin{cases} -1 & \text{if } \theta_i < 0 \\ 0 & \text{if } \theta_i = 0 \\ +1 & \text{if } \theta_i > 0 \end{cases} \tag{8}$$

Equation (8) is known as the subgradient vector of the Lasso cost function and is similar to Equation (4).

Elastic Net

Includes an additional parameter for regularization, given by r known as the mix ratio. If $r = 0$ elastic net is just Ridge Regression and if $r = 1$ elastic net is just Lasso Regression.

$$L(\boldsymbol{\theta}) = MSE(\boldsymbol{\theta}) + r\alpha \sum_{i=1}^{n} |\theta_i| + \frac{1-r}{2}\alpha \sum_{i=1}^{n} \theta_i^2 \tag{9}$$

Exercises

Answers to the questions in the Geron Textbook from Chapter 4.

1. If you have millions of features in a training set, it would make sense to use stochastic gradient descent in order to train the data for linear regression due to the amount of time and compute power it would require to train the data using any other method.
2. When the training data has very different scales, gradient descent methods might take longer to converge to a solution, the way to prevent this from happening is to scale the features before training them.

4 Support Vector Machines (SVM's)

This section covers mostly chapter 5 from the Geron book, and my MATH 4042U Notes. One of the most versatile machine learning algorithms, that can be used for classification both linear and non linear and can even be used fo regression. Linear vs Non Linear classification is describing the decision boundary and the shape that is used to determine classification.

The idea behind SVM's are to find a plane that has the maximum margin, or in other words the maximum distance between points of two data classes for classification. This plane is called a *hyperplane* in $\mathbb{R}^n$ that divides $\mathbb{R}^n$ into two half spaces.

$$\begin{aligned} H_+ &= \{(x_1, x_2, \ldots, x_n) \mid a_1x_1 + a_2x_2 + \ldots + a_nx_n > b\} \\ H_- &= \{(x_1, x_2, \ldots, x_n) \mid a_1x_1 + a_2x_2 + \ldots + a_nx_n < b\} \end{aligned}$$

SVM models maximize the margins from both groups, the algorithm learns the hyperplane whose distance to the nearest element of each group is largest, and this is called the maximal margin boundary.

Hard Margin Classification is used typically when the data is linearly separable, this type of classification is not quite good when dealing with outliers in our data, the decision boundary is very strict as opposed to *Soft Margin Classification* where we are trying to make the model as flexible as possible and are fine with some misclassifications but the idea is to have a balance between working with outliers and linearly separable data, so that the model can actually be deployed and be used on actual data. With Soft Margin we introduce a hyperparameter given by `C`, which controls how large or how small our margin will be.

In more real life situations, data will not be linearly separable, so we must embed our data points into a space so they can be separated by a hyperplane. For example, if we have a set of points (x, y), we can embed them into $\mathbb{R}^3$ by mapping the points to $(x, y, x^2 + y^2)$, so now we have introduced a third axis, which is given by the equation of a circle. This is called a *Kernel Trick*, and there are a variety of different ones built into scikit-learn that can be used in order to transform different data, i.e. linear and non linear.

Below is the SVM implemented on a penguins data set used in my MATH 4042U class.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn import svm
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import seaborn as sns


penguins = pd.read_csv('penguins.csv')
print(penguins.head())

# Plotting all attributes
sns.pairplot(penguins, hue='species')

# Applying SVM to the whole dataset
X = penguins.drop(columns=['species'])
y = penguins.loc[:, 'species']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

clf = svm.SVC(kernel='linear', C=1)
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
acc = accuracy_score(y_test, y_pred)
print(f"Accuracy: {acc}")
print(y_pred)

# Making prediction from data
backyard_prediction = clf.predict([[52, 15, 210, 6000]])[0]
print(f"The penguin species in the backyard is {backyard_prediction}")

# Applying SVM to data that looks linearly separable
X3 = penguins[["body_mass_g", "bill_length_mm"]]
y3 = penguins["species"]

X3_train, X3_test, y3_train, y3_test = train_test_split(X3, y3, test_size=0.2, random_state=1)

# Train SVM model
clf3 = svm.SVC(kernel='linear', C=1)
clf3.fit(X3_train, y3_train)
y3_pred = clf3.predict(X3_test)
accuracy3 = accuracy_score(y3_test, y3_pred)
print(f"Accuracy:{accuracy3}")
```

```
# Applying SVM with a different kernel
X8 = penguins[['bill_depth_mm', 'body_mass_g']]  # Select the right columns
y8 = penguins['species']  # Target variable

# Split into train and test sets
X8_train, X8_test, y8_train, y8_test = train_test_split(X8, y8, test_size=0.2, random_state=11)

# Train SVM model
clf8 = svm.SVC(kernel='sigmoid', C=1)
clf8.fit(X8_train, y8_train)
y8_pred = clf8.predict(X8_test)
accuracy8 = accuracy_score(y8_test, y8_pred)
print(f"Accuracy:{accuracy8}")
```

5 Dimensionality Reduction

There are many different way to do dimensionality reduction for a set of data, a lot of this section comes from my MATH 4042 class and some of the more advanced implementations come from Gerons book, like kernel PCA, etc.

Principal Component Analysis (PCA)

The whole idea of PCA is to be able to transform data points by centering the data, and then performing a series of operations to understand the direction that most of the variation is a dataset points towards, which allows us to understand the most important features that are part of a dataset, and could allow us to remove certain dimensions to the dataset that do no provide very valuable information and could help our program run faster or maybe even run more accurately by removing this noise.

Principal Components are orthogonal axes, that are used to capture the maximum amount of variance in the dataset, and tell us where most of the variation in the dataset comes from. Hence the name, Principal Component Analysis. Principal Components are new coordinate axes in the feature space and are linear combinations of the original features, the first PC captures the direction of the most variation in the data, the next PC's are all orthogonal to each other, meaning they form something called an Orthonormal Basis. The most common way to find these principal components is by finding the eigenvalues and the eigenvectors of data that has been centered around 0 and 1. The eigenvalues and its corresponding eigenvector tell us how well a Principal Component explains the variation of our data, it is also important to note again that each of these principal components are orthogonal to each other.

The first way we are going implement PCA into a dataset is by using the covariance matrix and eigenvalues and eigenvectors from a dataset. The steps are as followed for doing PCA through the use of *Eigen-decomposition.*

1. Take a Data Matrix A, and center the data using the equation, where μ is just the mean of the data, and $\pi^\top$ is just a row one 1's, which allow us to center our data and make the mean of the data be 0

$$B = A - \mu\pi^\top$$

2. Calculate the Covariance matrix given by

$$S = \frac{1}{n-1} B^\top B$$

3. From here is a much more tricky part, we need to find the eigenvalues and eigenvectors of S

4. Once that is complete we can divide each individual eigenvalue by the sum of all the eigenvalues found to find which eigenvectors explain most of the variance in the dataset, this is known as the Explained Variance Ratio

The eigenvectors are there to tell us the direction of the most variation in the dataset, and the eigenvalues are there to explain the amount of variance.

It is also important to note that regardless of the size of the dataset it is important to standardize the data, because say we have, 1 column of data with numbers ranging between 5000-10000 and another column of data that ranges from 10-33, then all that is going to happen if we do not standardize our data, is that the principal components are going to point toward the direction of the 5000-10000 ranging data points, because they are so large they render the smaller data points insignificant. So it is important in step 1 to also standardize the data, which you will see in the implementation section later.

The second way to implement PCA, is to use *singular value decomposition (SVD)*. We once again need to center our data like how we did in step 1 above. But we must decompose our matrix into 3 different matrices using equation (10).

$$B = U\Sigma V^\top \tag{10}$$

U is the left singular vectors, Σ is the matrix containing the singular values of the data matrix, which are always non zero, the singular values of the matrix are the square root of the non zero eigenvalues, and using these is how we are able to calculate the explained variance ratio. V is known as the right singular vectors of the data matrix and they are equal to the principal components of B, so in other words they tell us the direction of maximum variation in the dataset. To understand more intuitively what this means, V defines the new directions, while $U\Sigma$ is there to tell you where each of the data points sit on the new axes. U are our unit vectors for projecting data, Σ scales our data related to the variance and V is our new coordinate system. When we implement PCA using scikit-learn, SVD is used under the hood of the library.

The next way to implement PCA is by performing kernel PCA, which just utilizes different kernels in order to reduce dimensionality. Similar to the kernel trick like for SVM's, we can do the same thing for PCA to compute complex nonlinear projections that can be used for dimensionality reduction. We are unable to see how influential each attribute is, rather when we do kernel PCA we can inverse transform it back to the original data, and is sometimes used to find anomalies in the data once transformed back. Usually though, it is used for downstream input for a different model, such as a classification model.

The implementation below in python is using an MLB data set that is used in the Dive into Data Science book, the beginning of the code is cleaning up the data, and then the different PCA algorithms are implemented.

```python
import pandas as pd
import numpy as np
from sklearn.decomposition import PCA
from sklearn.decomposition import KernelPCA
from sklearn.impute import SimpleImputer

mlb = pd.read_csv("./Data/mlb.xls")
mlb = mlb.drop(['name', 'team', 'position'], axis=1)
print(mlb)


# Replace inf/-inf with NaN
mlb.replace([np.inf, -np.inf], np.nan, inplace=True)

# Create an imputer to fill NaNs with the mean of each column
imputer = SimpleImputer(strategy='mean')

# Apply imputer
mlb_imputed = pd.DataFrame(imputer.fit_transform(mlb), columns=mlb.columns)
print(mlb_imputed)

X_mlb = mlb_imputed['age'].values.reshape(-1,1)
y_mlb = mlb_imputed['weight'].values.reshape(-1,1)

# Check for invalid values
print(np.any(np.isnan(X_mlb)))
print(np.any(np.isinf(X_mlb)))
print(np.any(np.isnan(y_mlb)))
print(np.any(np.isinf(y_mlb)))

X_mlb_b = np.c_[np.ones((1034,1)), X_mlb]
print(X_mlb_b)

theta_best_mlb_2 = np.linalg.inv(X_mlb_b.T.dot(X_mlb_b)).dot(X_mlb_b.T).dot(y_mlb)
print(theta_best_mlb_2)

mlb_imputed_standard = (mlb_imputed - mlb_imputed.mean(axis=0)) / mlb_imputed.std(axis=0)
mlb_imputed_standard.cov()


# PCA using sklearn
pca_mlb = PCA(n_components=3)
pca_mlb.fit(mlb_imputed_standard)

print(pca_mlb.explained_variance_ratio_)
X_pca = pca_mlb.fit_transform(mlb_imputed_standard) # This fits PCA to X, then actually projects

component_names = [f"PC{i+1}" for i in range(X_pca.shape[1])]
 # Create loadings
loadings = pd.DataFrame(
    pca_mlb.components_.T,  # transpose the matrix of loadings
    columns=component_names,  # so the columns are the principal components
    index=mlb_imputed_standard.columns,  # and the rows are the original features
)

print(loadings) #produces eigenvectors that say how much each attribute contributes to the varian


# Kernel PCA using sklearn using an rbf kernel
rbf_pca = KernelPCA(n_components=3, kernel='rbf')

X_kpca = rbf_pca.fit_transform(mlb_imputed_standard)
lambdas = np.var(X_kpca, axis=0)
explained_variance_ratio = lambdas / np.sum(lambdas)
print(explained_variance_ratio)
print(rbf_pca.lambdas_)


evr = rbf_pca.lambdas_ / np.sum(rbf_pca.lambdas_)
print(evr)


# Kernel PCA using sklearn using a Sigmoid Kernel
sig_pca = KernelPCA(n_components=3, kernel='sigmoid')

X_sig_pca = sig_pca.fit_transform(mlb_imputed_standard)
lambdas_sig = np.var(X_sig_pca, axis=0)
evr_sig = lambdas_sig / np.sum(lambdas_sig)

print(evr_sig)
```

Locally Linearly Embedding (LLE)

A nonlinear dimensionality reduction technique that uses manifold learning as opposed to PCA which uses projections in order to perform dimensionality reduction.

Exercises

Solutions to the exercises in the Geron textbook.

1. The main motivation for reducing dimensionality of a dataset, is to remove redundant features, and reduce the size of a dataset, which improves the computational time complexity, but the drawback to this is that it may result in some lose of information along the way.

2. The *Curse of Dimensionality* is when a high dimensional data set, results in a model having very low generality.

3. If you performed dimensionality reduction in python, you can not get back your original data, unless you have previously saved it.

4. So we can perform PCA on a nonlinear dataset, but it is not ideal, because PCA works best for linear datasets, but we could try other techniques that would be more effective like kernel PCA or LLE, which perform better with non linear data.

6 Clustering Algorithms

This is the first time that we are now discussing algorithms that are associated with *Unsupervised Learning* the previous sections cover supervised learning involving regression or classification tasks. My professor said that clustering can be thought of as "a subset of stuff that have things in common" or "a subset of data points that have things in common". The problem faced in clustering is that we are given n data points we need to divide them into clusters, we want to group the data points such that in a single group, the distances between all points or the average distance between the points is small. The more informal definition for clustering I would say is grouping data points together based on "similarity".

K-Means

If we have a set of data points $(x_1, x_2, \dots, x_n)$ we are able to partition the data into k clusters, and the center of each cluster is known as the *centroid.* The idea is to minimize the chosen distance metric between the data point and the centroid in order to successfully partition our data points.

$$J = \sum_{i=1}^{k} \sum_{x_j \in S_i} \|x_j - \mu_i\|^2 \tag{11}$$

Equation (11) is the *objective function* for k-means clustering, which just means a mathematical function that an algorithm is trying to optimize. In the case of K-means it is trying to minimize the ℓ_2 norm between the data point and the centroid. The steps for the k-means algorithm is as followed.

1. Begin by choosing random points k, which are going to be our centroids represented by μ_k, where k is the amount of centroids we have chosen for our k-means algorithm, this is the initialization of the algorithm

2. The next step we must assign each data point x_j to a cluster, by calculating the distance between the data point and each centroid and the data point is assigned to the cluster of the centroid that it is closest to, usually the distance metric used is the Euclidean norm (also known as the ℓ_2 norm)

$$S_i^{(t)} = \{x_j \mid \|x_j - \mu_i^{(t)}\|^2 \leq \ \|x_j - \mu_p^{(t)}\|^2,\ \forall p = 1, \dots, p\} \tag{12}$$

3. Recalculate the centroids location, by factoring in the new data point added to the cluster

$$\mu_i^{(t+1)} = \frac{1}{|S_i^{(t)}|} \sum_{x \in S_i^{(t)}} x \tag{13}$$

We are recalculating the centroid for the future step, by summing over all the data points in the cluster and dividing by the number of data points in the cluster

4. Loop through steps 2 and 3 until convergence.

You can also use different distance metrics and different norms to compute the distance between data points and the centroid, its just the most common and the one used in naive k-means is usually Euclidean distance, we do this because the mean is the point that minimizes the sum of squared distances.

Spectral Clustering

This method of finding clusters in data makes great use of linear algebra to find clusters by associating a graph with out data. Once that is done, from the eigenvalues and eigenvectors of the matrix we are able to work with the graph.

7 Artificial Neural Networks (ANN)

A *Perceptron* is the simplest ANN architecture, and is based on something called a threshold logic unit (TLU), the simplest version is a *single layer perceptron* which is also known as a *single layered neural network* and is typically used for classification. The inputs of a perceptron is numbers as well as the outputs. Each input is associated with some weight, and the TLU computes the weighted sum of all the inputs which is in other words a linear combination of the inputs, then it applies a step function to that sum which then outputs the result. The step function is typically, either a Heaviside function or a sign function, and is very effective with data that looks linearly separable.

$$h_{\boldsymbol{W},\boldsymbol{b}}(\boldsymbol{X}) = \phi(\boldsymbol{X}\boldsymbol{W} + \boldsymbol{b}) \tag{14}$$

Equation (14) computes the outputs of a layer of the artificial neurons for all the instances at once using linear algebra. When all the neurons in a layer are connected to every neuron in the previous layer, the layer is called a *fully connected layer*.

Perceptrons are trained by following *Hebb's Rule*, or *Hebbian Learning*, based on the idea that when one neuron fires the another neuron does as well and the connection between the two neurons grow stronger. Perceptrons are trained by taking in account the error when producing a prediction, Hebb's rule reinforces the connections in order to reduce the error. Perceptrons learning rule, or the weight update of the perceptron is given by equation (12).

$$w_{i,j}^{(next)} = w_{i,j} + \eta(y_j - \hat{y}_j)x_i \tag{15}$$

When the output neuron produces a prediction that is wrong, the learning rule, reinforces the connection weights with the predictions that would have contributed to the prediction being correct. The weight matrix $\boldsymbol{W}$ entries are initially chosen randomly, but are then corrected as said before to improve accuracy. The equation (15) is quite similar to that of gradient descent techniques used earlier, and includes η in the same way, as the learning rate. Single Layered perceptrons, are linear classifiers so unlike logistic regression they cannot output a class probability, they can only output either a 1 or a 0, it is either on/off.

Multilayer Perceptron (MLPs)

Composed of at minimum three layers. The first layer is the *Input Layer*, there is only one of these. The next layer of TLUs is called the *Hidden Layer*, which there can be multiple of. The final later is the *Output Layer.* The input and hidden layers are the only ones that contain a bias neuron and they are fully connected to the next layer.

The *Backpropagation Algorithm* is what is used to train these MLPs. It computes the gradient of the neural networks error with respect to every single parameter in the model. In simpler terms, it is able to compute how much each bias term and connection should be tweaked in order to reduce the amount of error in the model. After the gradients are found, it performs a regular gradient descent step, and this process is repeated until the neural network converges to a solution. MLPs can be used for multi-label classification and multivariate regression tasks unlike a single layered perceptron.

8 Linear Algebra & Statistics Topics

There are tons of Linear Algebra, Statistical Methods and Probability Methods that should be fully understood, to have a full grasp of machine learning, and why certain things work numerically. Below you will find a list of topics that could be studied further or are already included in part in these notes.

1. Linear Algebra
    - Singular Value Decomposition
    - Alternating Least Squares (Used in Recommender Systems)
    - Eckhart-Young Theorem
    - Fiedler Vectors
    - Total Least Squares
2. Statistical Methods & Probability
    - Central Limit Theorem
    - Bayesian Statistics
    - Bayesian Regression
    - Maximum Likelihood Estimations
    - Joint Probability Distributions
    - Hypothesis Testing
    - Confidence Intervals
    - ANOVA, ANCOVA
3. Discrete Mathematics
    - Counting/Combinatorics
    - Recursion
    - Graph Theory/Network Science

9 Things to still add

- For the Normal Equation, add shapes of matrices in description.
- Explain what each matrix is in single layed perceptron