FIRST EDITION

PYTHON MACHINE LEARNING

SUPERVISED BY
AMIRSINA TORFI

MACHINE LEARNING
BASICS

Machine-Learning-Course Documentation

Release 1.0

Amirsina Torfi

Nov 13, 2019

Table of Content

CHAPTER 1

Authorship

Creator: Machine Learning Mindset [Blog, GitHub, Twitter]

Supervisor: Amirsina Torfi [GitHub, Personal Website, Linkedin]

Developers: Brendan Sherman*, James E Hopkins* [Linkedin], Zac Smith [Linkedin]

NOTE: This project has been developed as a capstone project offered by [CS 4624 Multimedia/ Hypertext course at Virginia Tech] and Supervised and supported by [Machine Learning Mindset].

*: equally contributed

CHAPTER 2

Introduction

The purpose of this project is to provide a comprehensive and yet simple course in Machine Learning using Python.

2.1 Machine Learning Overview

2.1.1 How was the advent and evolution of machine learning?

You can argue that the start of modern machine learning comes from Alan Turing's "Turing Test" of 1950. The Turing Test aimed to find out if a computer is brilliant (or at least smart enough to fool a human into thinking it is). Machine learning continued to develop with game playing computers. The games these computers play have grown more complicated over the years from checkers to chess to Go. Machine learning was also used to model pattern recognition systems in nature such as neural networks. But machine learning didn't just stay confined to large computers stuck in rooms. Robots were designed that could use machine learning to navigate around obstacles automatically. We continue to see this concept in the self-driving cars of today. Machine learning eventually began to be used to analyze large sets of data to conclude. This allowed for humans to be able to digest large, complex systems through the use of machine learning. This was an advantageous result for those involved in marketing and advertisement as well as those concerned with complex data. Machine learning was also used for image and video recognition. Machine learning allowed for the classification of objects in pictures and videos as well as identification of specific landmarks of interest. Machine learning tools are now available through the Cloud and on large scale distributed systems.

2.1.2 Why is machine learning important?

Machine learning has practical applications for a range of common business problems. By using machine learning, organizations can complete tasks in less time and more efficiently. One example could be preprocessing a set of data for a future stage that requires human intervention. Tasks that would have previously required lots of user input can now be automated to some degree. The saved resources can then be put towards something else that needs to be done. Beyond task automation, machine learning can be used to analyze large quantities of complex data to make predictions. Data analysis is an essential task for many businesses. For example, a company could analyze sales data to find out where profitable opportunities are or to find out where it risks losing money. Using machine learning can potentially allow for real-time analysis of complex data. Such an ability might be required for mission-critical systems. Machine

learning is also an important topic for research and continued development. Currently, machine learning still has a lot of limitations and isn't close to replacing the need for a live person. Machine learning's constant evolution could offer solutions for hard problems that might take up too many resources now to even consider.

2.1.3 Who is using ML and why (government, healthcare system, etc.)?

Machine learning stands to impact most industries in some way so many managers and higher-ups are trying to at least learn what it is if not what it can do for them. Machine learning models are expected to get better at prediction when supplied with more information. Nowadays, it is effortless to obtain large amounts of information that can be used to train very accurate models. The computers of today are also stronger than those available in the past and offer options such as cloud solutions and distributed processing to tackle hard machine learning problems. Many of these options are readily available to almost anyone can use machine learning. We can see examples of machine learning in self-driving cars, recommendation systems, linguistic analysis, credit scoring, and security to name a few. Financial services can use machine learning to provide insights about client data and to predict areas of risk. Government agencies with access to large quantities of data and an interest in streamlining or at least speeding up parts of their services can utilize machine learning. Health care providers with cabinets full of patient data can use machine learning to aid in diagnosis as well as identifying health risks. Shopping services can use customers' purchase histories and machine learning techniques to make personalized recommendations and gauge dangerous products. Anyone with a large amount of data stands to profit from using machine learning.

2.1.4 Further Reading

- https://www.forbes.com/sites/bernardmarr/2016/02/19/a-short-history-of-machine-learning-every-manager-should-read/#15f4059615e7
- https://www.interactions.com/blog/technology/machine-learning-important/
- https://www.sas.com/en_us/insights/analytics/machine-learning.html
- https://www.simplilearn.com/what-is-machine-learning-and-why-it-matters-article

CHAPTER 3

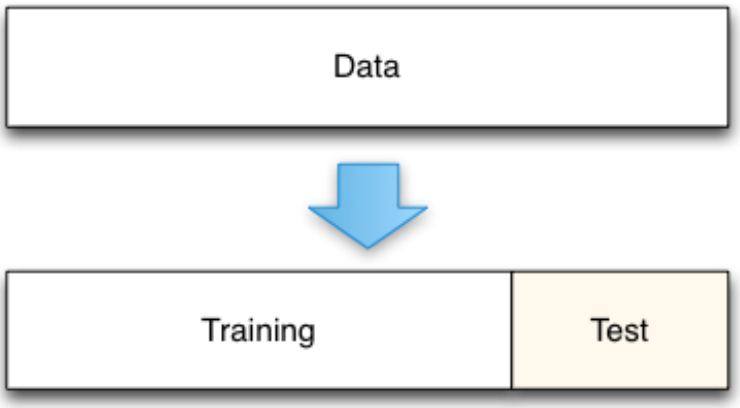
Cross-Validation

3.1 Motivation

It's easy to train a model against a particular dataset, but how does this model perform when introduced with new data? How do you know which machine learning model to use? Cross-validation answers these questions by assuring a model is producing accurate results and comparing those results against other models. Cross-validation goes beyond regular validation, the process of analyzing how a model does on its own training data, by evaluating how a model does on *new* data. Several different methods of cross-validation are discussed in the following sections:

3.2 Holdout Method

The holdout cross-validation method involves removing a certain portion of the training data and using it as test data. The model is first trained against the training set, then asked to predict output from the testing set. This is the simplest form of cross-validation techniques, and is useful if you have a large amount of data or need to implement validation quickly and easily.



Typically the holdout method involves splitting a dataset into 20-30% test data and the rest as training data. These numbers can vary - a larger percentage of test data will make your model more prone to errors as it has less training experience, while a smaller percentage of test data may give your model an unwanted bias towards the training data. This lack of training or bias can lead to Underfitting/Overfitting of our model.

3.3 K-Fold Cross Validation

K-Fold Cross Validation helps remove these biases from your model by repeating the holdout method on k subsets of your dataset. With K-Fold Cross Validation, a dataset is broken up into several unique folds of test and training data. The holdout method is performed using each combination of data, and the results are averaged to find a total error estimation.

A "fold" here is a unique section of test data. For instance, if you have 100 data points and use 10 folds, each fold contains 10 test points. K-Fold Cross Validation is important because it allows you to use your complete dataset for both training and testing. It's especially useful when evaluating a model using small or limited datasets.

3.4 Leave-P-Out / Leave-One-Out Cross Validation

Leave-P-Out Cross Validation (LPOCV) tests a model by using every possible combination of P test data points on a model. As a simple example, if you have 4 data points and use 2 test points, the model will be trained and tested as follows:

```
1: [ T T - - ]
2: [ T - T - ]
3: [ T - - T ]
4: [ - T T - ]
5: [ - T - T ]
6: [ - - T T ]
```

Where "T" is a test point, and "-" is a training point. Below is another visualization of LPOCV:

LPOCV can provide an extremely accurate error estimation, but can quickly become exhaustive for large datasets. The amount of testing iterations a model has to go through using LPOCV can be calculated using a mathematical combination n C P, with n being our total number of data points. We can see, for instance, that a LPOCV run using a dataset of 10 points with 3 test points would require 10 C 3 = 120 iterations.

Fig. 1: Ref: http://www.ebc.cat/2017/01/31/cross-validation-strategies/

Because of this, Leave-One-Out Cross Validation (LOOCV) is a commonly used cross-validation method. It is just a subset of LPOCV, with P being 1. This allows us to evaluate a model in the same number of steps as there are data points. LOOCV can also be seen as K-Fold Cross Validation, where the number of folds is equal to the number of data points.

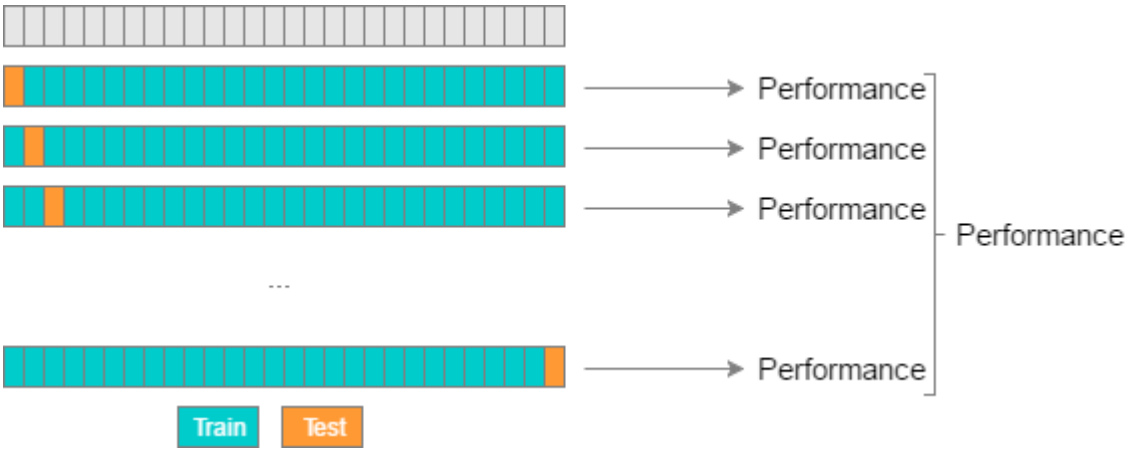


Fig. 2: Ref: http://www.ebc.cat/2017/01/31/cross-validation-strategies/

Similar to K-Fold Cross Validation, LPOCV and LOOCV train a model using the full dataset. They are particularly useful when you're working with a small dataset, but incur performance tradeoffs.

3.5 Conclusion

Cross-validation is a way to validate your model against new data. The most effective forms of cross-validation involve repeatedly testing a model against a dataset until every point or combination of points have been used to validate a model, though this comes with performance trade-offs. We discussed several methods of splitting a dataset for cross-validation:

- Holdout Method: Splitting a percent of data off as test data
- K-Fold Method: Dividing data into sections, using each as a test/train split
- Leave-P-Out Method: Using every combination of a number of points (P) as test data

3.6 Motivation

There are many different types of machine learning models, including Linear/Logistic Regression, K-Nearest-Neighbors, and Support Vector Machines - but how do we know which type of model is the best for our dataset? Using a model unsuitable for our data will lead to less accurate predictions, and could lead to financial, physical, or other forms of harm. Individuals and companies should make sure to cross-validate any models they put into use.

3.7 Code Examples

The provided code shows how to split a set of data with the three discussed methods of cross-validation using Scikit-Learn, a Python machine learning library.

holdout.py splits a set of sample diabetes data using the Holdout Method. In scikit-learn, this is done using a function called `train_test_split()` which randomly splits a set of data into two portions:

```
TRAIN_SPLIT = 0.7
...

dataset = datasets.load_diabetes()
...

x_train, x_test, y_train, y_test = train_test_split(...)
```

Note that you can change the portion of data used for training by changing the `TRAIN_SPLIT` value at the top. This should be a number from 0 to 1. Output from this file shows the number of training and test points used for the split. It may be beneficial to see the actual data points - if you would like to see these, uncomment the last two print statements in the script.

---

k-fold.py splits a set of data using the K-Fold Method. This is done by creating a KFold object initialized with the number of splits to use. Scikit-learn makes it easy to split data by calling KFold's `split()` method:

```
NUM_SPLITS = 3
data = numpy.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12]])

kfold = KFold(n_splits=NUM_SPLITS)
split_data = kfold.split(data)
```

The return value of this is an array of train and test points. Note that you can play with the number of splits by changing the associated value at the top of the script. This script not only outputs the train/test data, but also outputs a nice bar where where you can track the progress of the current fold:

```
[ T T - - - - ]
Train: (2: [5 6]) (3: [7 8]) (4: [ 9 10]) (5: [11 12])
Test:  (0: [1 2]) (1: [3 4])
...
```

---

leave-p-out.py splits a set of data using both the Leave-P-Out and Leave-One-Out Methods. This is done by creating LeavePOut/LeaveOneOut objects, the LPO initialized with the number of splits to use. Similar to KFold, the train-test data split is created with the `split()` method:

```
P_VAL = 2
data = numpy.array([[1, 2], [3, 4], [5, 6], [7, 8]])

loocv = LeaveOneOut()
lpocv = LeavePOut(p=P_VAL)

split_loocv = loocv.split(data)
split_lpocv = lpocv.split(data)
```

Note that you can change the P value at the top of the script to see how different values operate.

3.8 References

1. https://towardsdatascience.com/cross-validation-in-machine-learning-72924a69872f

2. https://machinelearningmastery.com/k-fold-cross-validation/
3. https://www.quora.com/What-is-cross-validation-in-machine-learning
4. http://www.ebc.cat/2017/01/31/cross-validation-strategies/

CHAPTER 4

Linear Regression

4.1 Motivation

When we are presented with a data set, we try and figure out what it means. We look for connections between the data points and see if we can find any patterns. Sometimes those patterns are hard to see so we use code to help us find them. There are lots of different patterns data can follow so it helps if we can narrow down those options and write less code to analyze them. One of those patterns is a linear relationship. If we can find this pattern in our data, we can use the linear regression technique to analyze it.

4.2 Overview

Linear regression is a technique used to analyze a **linear relationship** between **input** variables and a single **output** variable. A **linear relationship** means that the data points tend to follow a straight line. **Simple linear regression** involves only a single input variable. *Figure 1* shows a data set with a linear relationship.

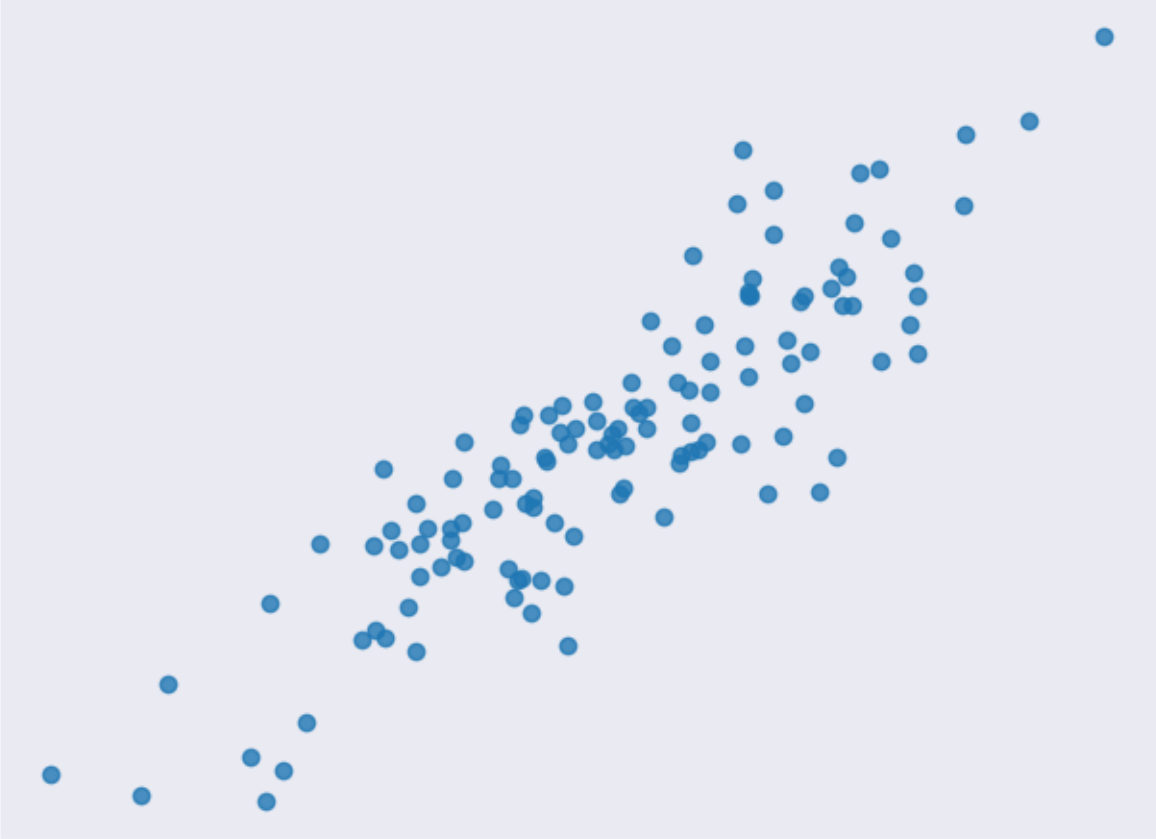


Fig. 1: **Figure 1. A sample data set with a linear relationship** [code]

Our goal is to find the line that best models the path of the data points called a line of best fit. The equation in *Equation 1*, is an example of a linear equation.

$$y = a_0 + a_1 x$$

Fig. 2: **Equation 1. A linear equation**

Figure 2 shows the data set we use in *Figure 1* with a line of best fit through it.

Let's break it down. We already know that x is the input value and y is our predicted output. a_0 and a_1 describe the shape of our line. a_0 is called the **bias** and a_1 is called a **weight**. Changing a_0 will move the line up or down on the plot and changing a_1 changes the slope of the line. Linear regression helps us pick appropriate values for a_0 and a_1.

Note that we could have more than one input variable. In this case, we call it **multiple linear regression**. Adding extra input variables just means that we'll need to find more weights. For this exercise, we will only consider a simple linear regression.

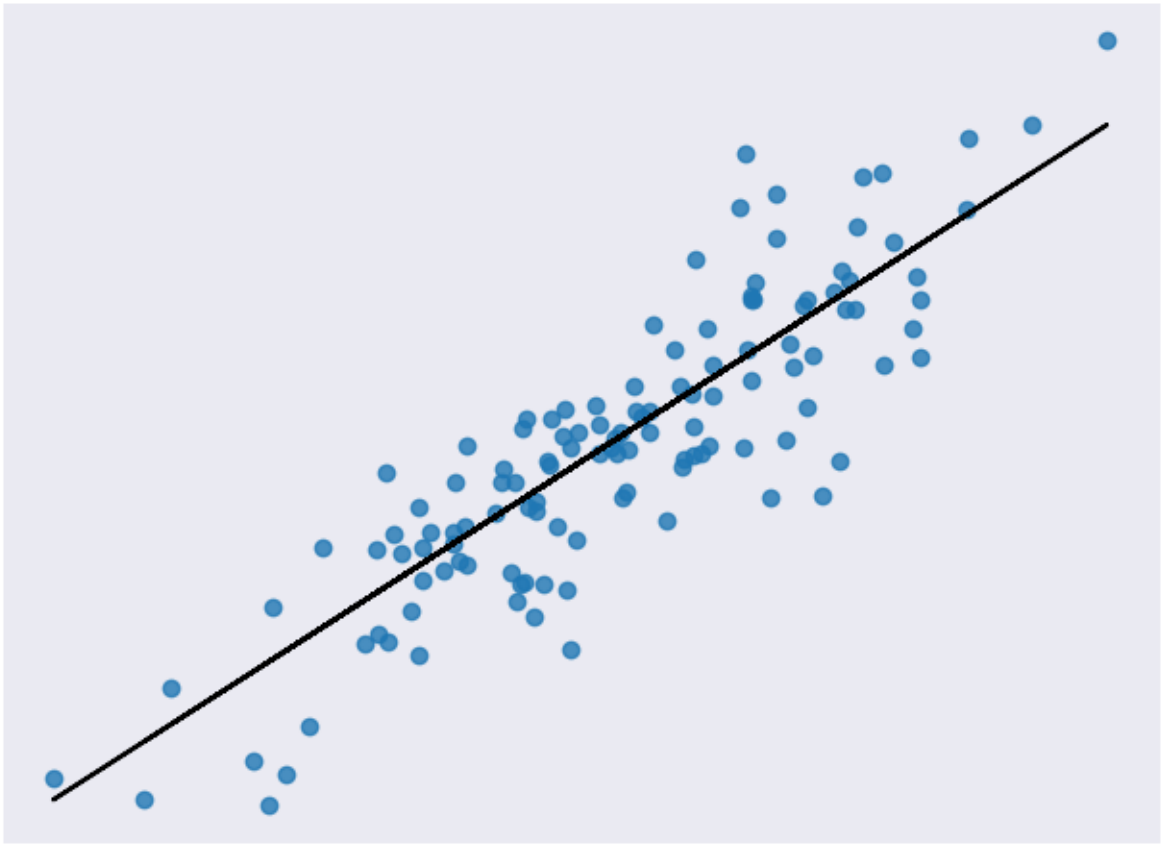


Fig. 3: **Figure 2. The data set from Figure 1 with a line of best fit** [code]

4.3 When to Use

Linear regression is a useful technique but isn't always the right choice for your data. Linear regression is a good choice when there is a linear relationship between your independent and dependent variables and you are trying to predict continuous values [*Figure 1*].

It is not a good choice when the relationship between independent and dependent variables is more complicated or when outputs are discrete values. For example, *Figure 3* shows a data set that does not have a linear relationship so linear regression would not be a good choice.

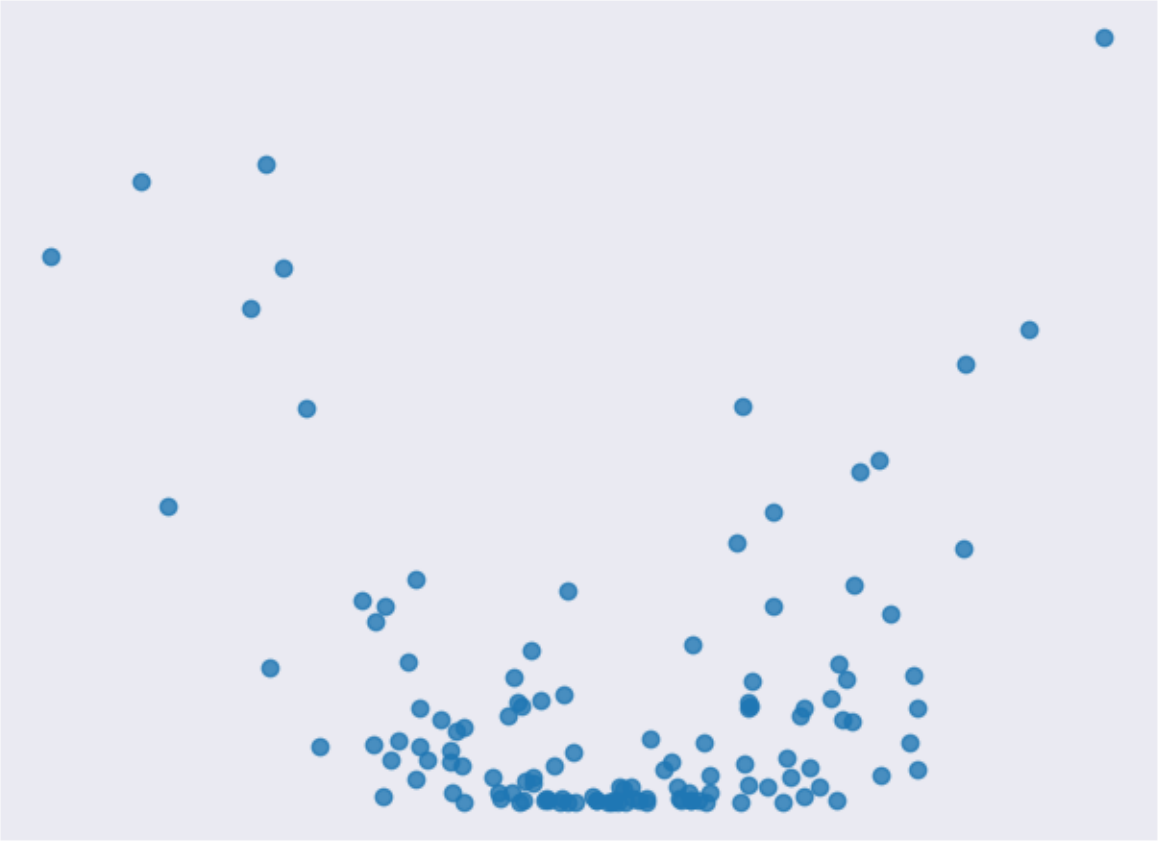


Fig. 4: **Figure3. A sample data set without a linear relationship** [code]

It is worth noting that sometimes you can apply transformations to data so that it appears to be linear. For example, you could apply a logarithm to exponential data to flatten it out. Then you can use linear regression on the transformed data. One method of transforming data in `sklearn` is documented here.

Figure 4 is an example of data that does not look linear but can be transformed to have a linear relationship.

Figure 5 is the same data after transforming the output variable with a logarithm.

4.4 Cost Function

Once we have a prediction, we need some way to tell if it's reasonable. A **cost function** helps us do this. The cost function compares all the predictions against their actual values and provides us with a single number that we can use

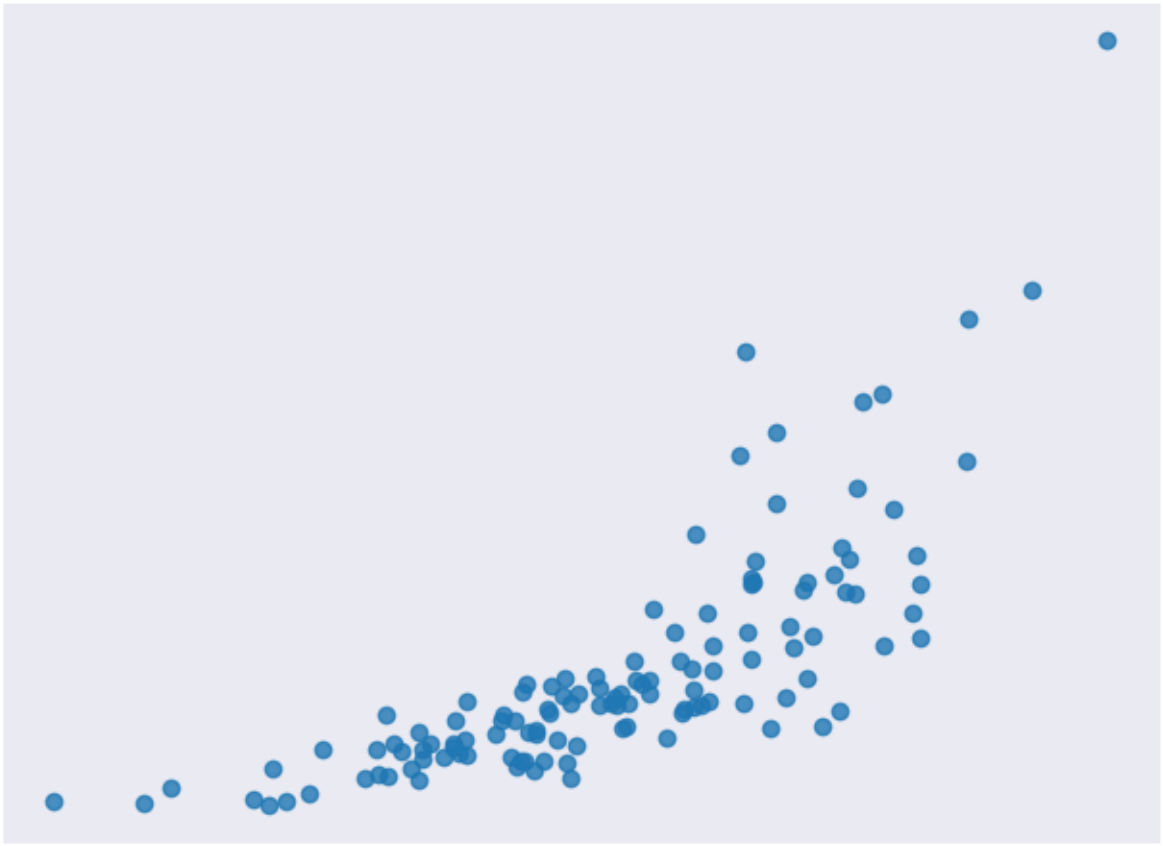


Fig. 5: **Figure 4. A sample data set that follows an exponential curve** [code]

Fig. 6: **Figure 5. The data set from Figure 4 after applying a logarithm to the output variable** [code]

to score the prediction function. *Figure 6* shows the cost for one such prediction.

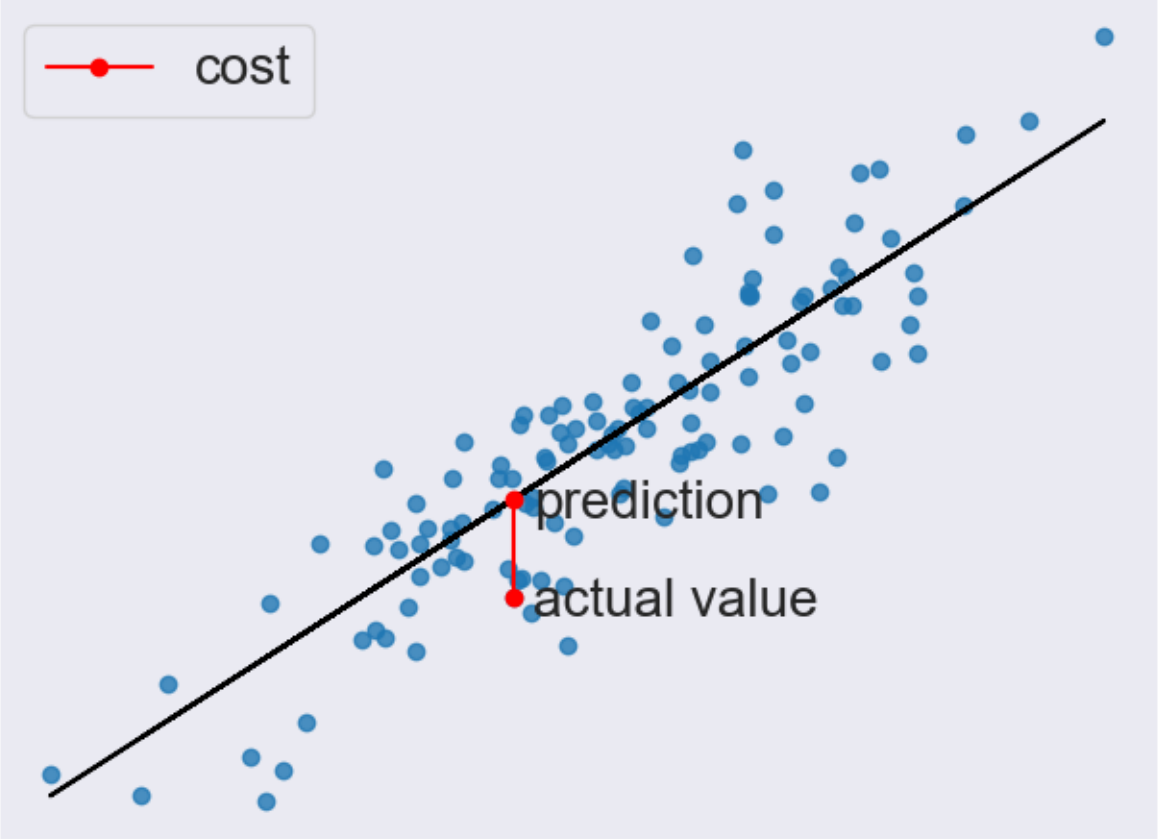


Fig. 7: **Figure 6. The plot from Figure 2 with the cost of one prediction emphasized** [code]

Two common terms that appear in cost functions are the **error** and **squared error**. The error [*Equation 2*] is how far away from the actual value our prediction is.

$$E = \text{prediction} - \text{actual}$$

Fig. 8: **Equation 2. An example error function**

Squaring this value gives us a useful expression for the general error distance as shown in *Equation 3*.

$$E^2 = (\text{prediction} - \text{actual})^2$$

Fig. 9: **Equation 3. An example squared error function**

We know an error of 2 above the actual value and an error of 2 below the actual value should be about as bad as each other. The squared error makes this clear because both of these values result in a squared error of 4.

We will use the Mean Squared Error (MSE) function shown in *Equation 4* as our cost function. This function finds the average squared error value for all of our data points.

Cost functions are important to us because they measure how accurate our model is against the target values. Making sure our models are accurate will remain a key theme throughout later modules.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(prediction_i - actual_i)^2$$

Fig. 10: **Equation 4. The Mean Squared Error (MSE) function**

4.5 Methods

A lower cost function means a lower average error across the data points. In other words, lower cost means a more accurate model for the data set. We will briefly mention a couple of methods for minimizing the cost function.

4.5.1 Ordinary Least Squares

Ordinary least squares is a common method for minimizing the cost function. In this method, we treat the data as one big matrix and use linear algebra to estimate the optimal values of the coefficients in our linear equation. Luckily, you don't have to worry about doing any linear algebra because the Python code handles it for you. This also happens to be the method used for this modules code.

Below are the relevant lines of Python code from this module related to ordinary least squares.

```
# Create a linear regression object
regr = linear_model.LinearRegression()
```

4.5.2 Gradient Descent

Gradient descent is an iterative method of guessing the coefficients of our linear equation in order to minimize the cost function. The name comes from the concept of gradients in calculus. Basically this method will slightly move the values of the coefficients and monitor whether the cost decreases or not. If the cost keeps increasing over several iterations, we stop because we've probably hit the minimum already. The number of iterations and tolerance before stopping can both be chosen to fine tune the method.

Below are the relevant lines of Python code from this module modified to use gradient descent.

```
# Create a linear regression object
regr = linear_model.SGDRegressor(max_iter=10000, tol=0.001)
```

4.6 Code

This module's main code is available in the linear_regression_lobf.py file.

All figures in this module were created with simple modifications of the linear_regression.py code.

In the code, we analyze a data set with a linear relationship. We split the data into a training set to train our model and a testing set to test its accuracy. You may have guessed that the model used is based on linear regression. We also display a nice plot of the data with a line of best fit.

4.7 Conclusion

In this module, we learned about linear regression. This technique helps us model data with linear relationships. Linear relationships are fairly simple but still show up in a lot of data sets so this is a good technique to know. Learning about

linear regression is a good first step towards learning more complicated analysis techniques. We will build on a lot of the concepts covered here in later modules.

4.8 References

1. https://towardsdatascience.com/introduction-to-machine-learning-algorithms-linear-regression-14c4e325882a
2. https://machinelearningmastery.com/linear-regression-for-machine-learning/
3. https://ml-cheatsheet.readthedocs.io/en/latest/linear_regression.html
4. https://machinelearningmastery.com/implement-simple-linear-regression-scratch-python/
5. https://medium.com/analytics-vidhya/linear-regression-in-python-from-scratch-24db98184276
6. https://scikit-learn.org/stable/auto_examples/linear_model/plot_ols.html
7. https://scikit-learn.org/stable/modules/generated/sklearn.compose.TransformedTargetRegressor.html

CHAPTER 5

Overfitting and Underfitting

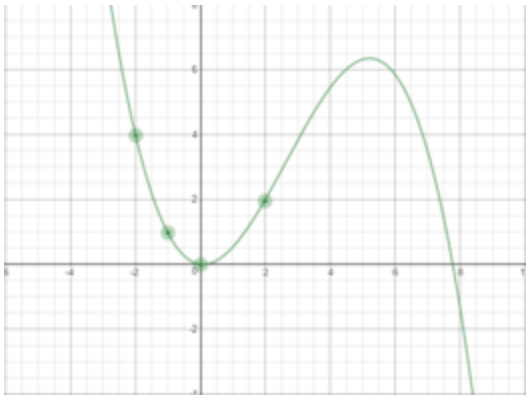
5.1 Overview

When using machine learning, there are many ways to go wrong. Some of the most common issues in machine learning are **overfitting** and **underfitting**. To understand these concepts, let's imagine a machine learning model that is trying to learn to classify numbers, and has access to a training set of data and a testing set of data.

5.2 Overfitting

A model suffers from **Overfitting** when it has learned too much from the training data, and does not perform well in practice as a result. This is usually caused by the model having too much exposure to the training data. For the number classification example, if the model is overfit in this way, it may be picking up on tiny details that are misleading, like stray marks as an indication of a specific number.

The estimate looks pretty good when you look at the middle of the graph, but the edges have large error. In practice, this error isn't always at edge cases and can pop up anywhere. The noise in training can cause error as seen in the graph below.



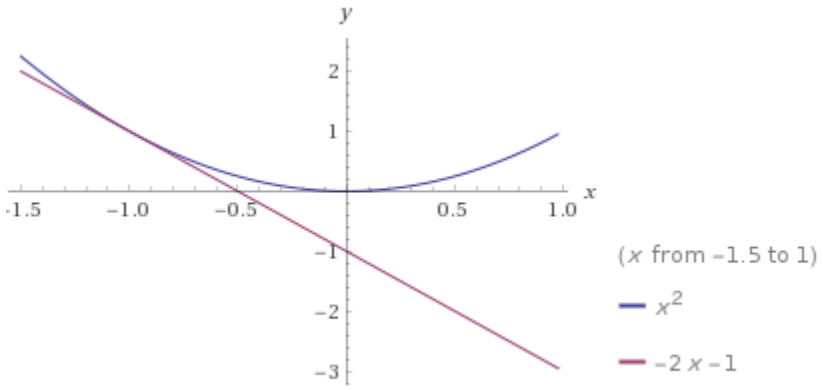
(Created using https://www.desmos.com/calculator/dffnj2jbow)

In this example, the data is overfit by a polynomial degree. The points indicated are true to the function y = x^2, but does not approximate the function well outside of those points.

5.3 Underfitting

A model suffers from **Underfitting** when it has not learned enough from the training data, and does not perform well in practice as a result. As a direct contrast to the previous idea, this issue is caused by not letting the model learn enough from training data. In the number classification example, if the training set is too small or the model has not had enough attempts to learn from it, then it will not be able to pick out key features of the numbers.

The issue with this estimate is clear to the human eye, the model should be nonlinear, and is instead just a simple line. In machine learning, this could be a result of underfitting, the model has not had enough exposure to training data to adapt to it, and is currently in a simple state.



(Created using Wolfram Alpha)

5.4 Motivation

Finding a good fit is one of the central problems in machine learning. Gaining a good grasp of how to avoid fitting problems before even worrying about specific methods can keep models on track. The mindset of hunting for a good fit, rather than throwing more learning time at a model is very important to have.

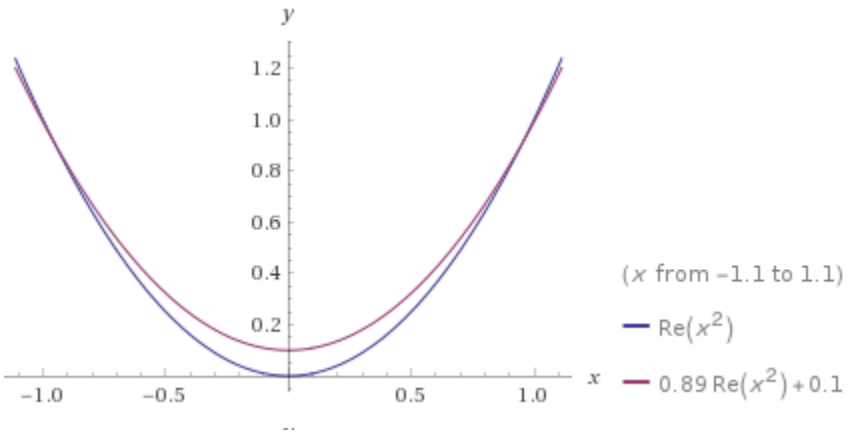
5.5 Code

The example code for overfitting shows some basic examples based in polynomial interpolation, trying to find the equation of a graph. The overfitting.py file, you can see that there is a true function being modeled, as well as some estimates that are shown to not be accurate.

The estimates are representations of overfitting and underfitting. For overfitting, a higher degree polynomial is used (x cubed instead of squared). While the data is relatively close for the chosen points, there are some artifacts outside of them. The example of underfitting, however, does not even achieve accuracy at many of the points. Underfitting is similar to having a linear model when trying to model a quadratic function. The model does well on the point(s) it trained on, in this case the point used for the linear estimate, but poorly otherwise.

5.6 Conclusion

Check out the cross-validation and regularization sections for information on how to avoid overfitting in machine learning models. Ideally, a good fit looks something like this:



(Created using Wolfram Alpha)

When using machine learning in any capacity, issues such as overfitting frequently come up, and having a grasp of the concept is very important. The modules in this section are among the most important in the whole repository, since regardless of the implementation, machine learning always includes these fundamentals.

5.7 References

1. https://machinelearningmastery.com/overfitting-and-underfitting-with-machine-learning-algorithms/
2. https://medium.com/greyatom/what-is-underfitting-and-overfitting-in-machine-learning-and-how-to-deal-with-it-6803a989c76
3. https://towardsdatascience.com/overfitting-vs-underfitting-a-conceptual-explanation-d94ee20ca7f9

CHAPTER 6

Regularization

6.1 Motivation

Consider the following scenario. You are making a peanut butter sandwich and are trying to adjust ingredients so that it has the best taste. You might consider the type of bread, type of peanut butter, or peanut butter to bread ratio in your decision making process. But would you consider other factors like how warm it is in the room, what you had for breakfast, or what color socks you're wearing? You probably wouldn't as these things don't have as much impact on the taste of your sandwich. You would focus more on the first few features for whatever recipe you end up using and avoid paying too much attention to the other ones. This is the basic idea of **regularization**.

6.2 Overview

In previous modules, we have seen prediction models trained on some sample set and scored against how close they are to a test set. We obviously want our models to make predictions that are accurate but can they be too accurate? When we look at a set of data, there are two main components: the underlying pattern and noise. We only want to match the pattern and not the noise. Consider the figures below that represent quadratic data. *Figure 1* uses a linear

model to approximate the data. *Figure 2* uses a quadratic model to approximate the data. *Figure 3* uses a high degree polynomial model to approximate the data.

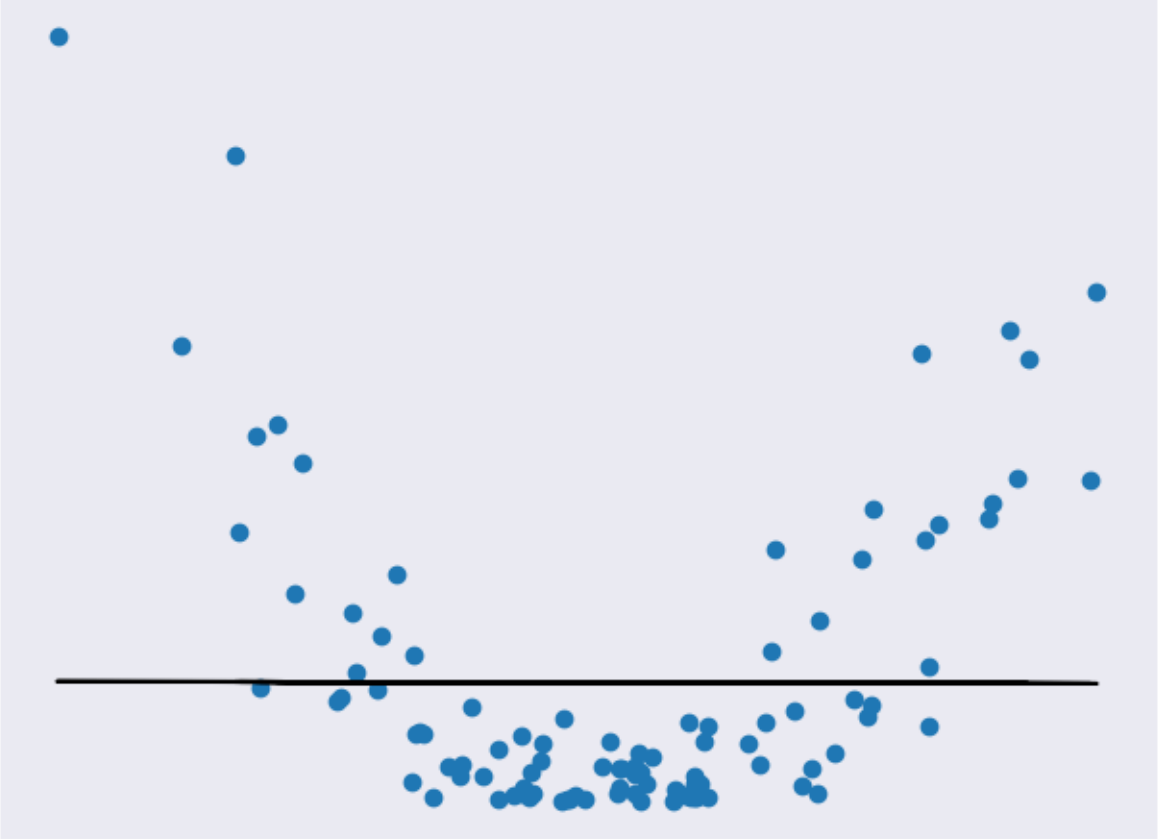


Fig. 1: **Figure 1. A linear prediction model** [code]

Figure 1 underfits the data, *Figure 2* looks to be a pretty good fit for the data, and *Figure 3* is a very close fit for the data. Of all the models above, the third is likely the most accurate for the test set. But this isn't necessarily a good thing. If we add in some more test points, we'll likely find that the third model is no longer as accurate at predicting them but the second model is still pretty good. This is because the third model suffers from overfitting. Overfitting means it does a really good job at fitting the test data (including noise) but is bad at generalizing to new data. The second model is a nice fit for the data and is not so complex that it won't generalize.

The goal of regularization is to avoid overfitting by penalizing more complex models. The general form of regularization involves adding an extra term to our cost function. So if we were using a cost function CF, regularization might lead us to change it to CF + λ * R where R is some function of our weights and λ is a tuning parameter. The result is that models with higher weights (more complex) get penalized more. The tuning parameter basically lets us adjust the regularization to get better results. The higher the λ the less impact the weights have on the total cost.

6.3 Methods

There are many methods we can use for regularization. Below we'll cover some of the more common ones and when they are good to use.

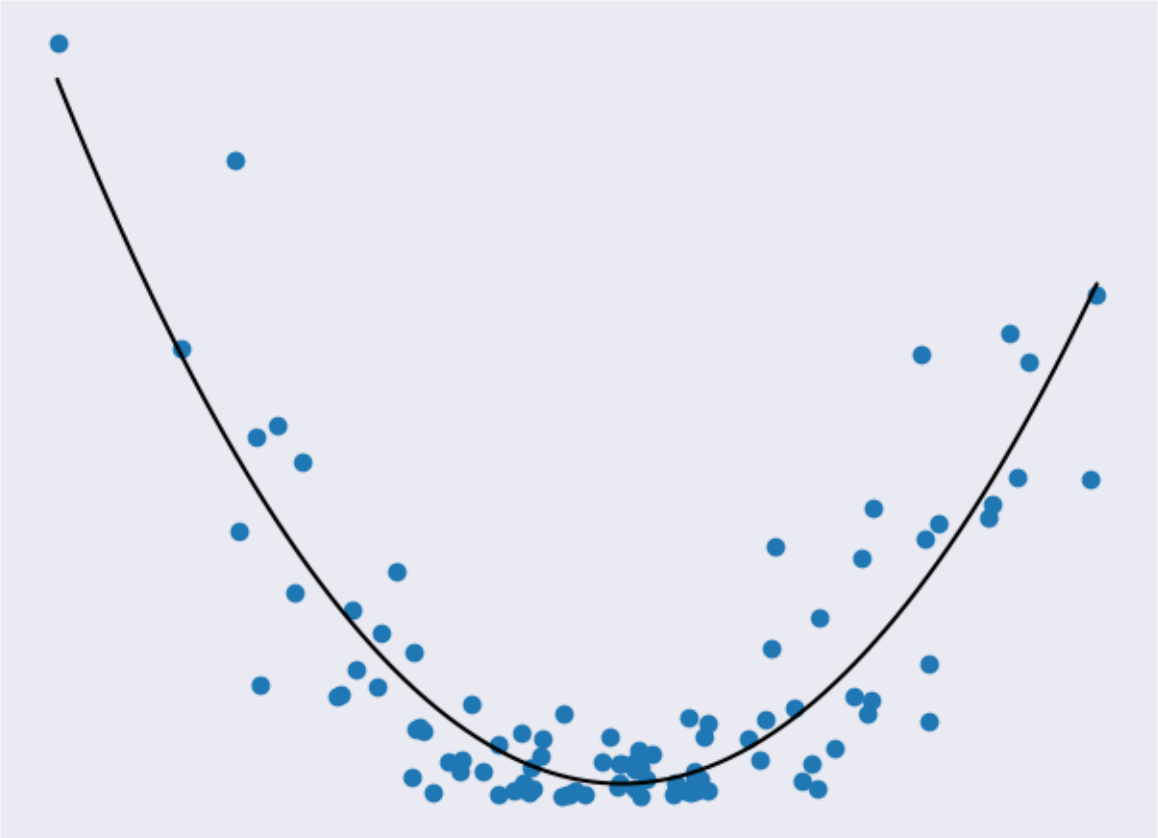


Fig. 2: **Figure 2. A quadratic prediction model** [code]

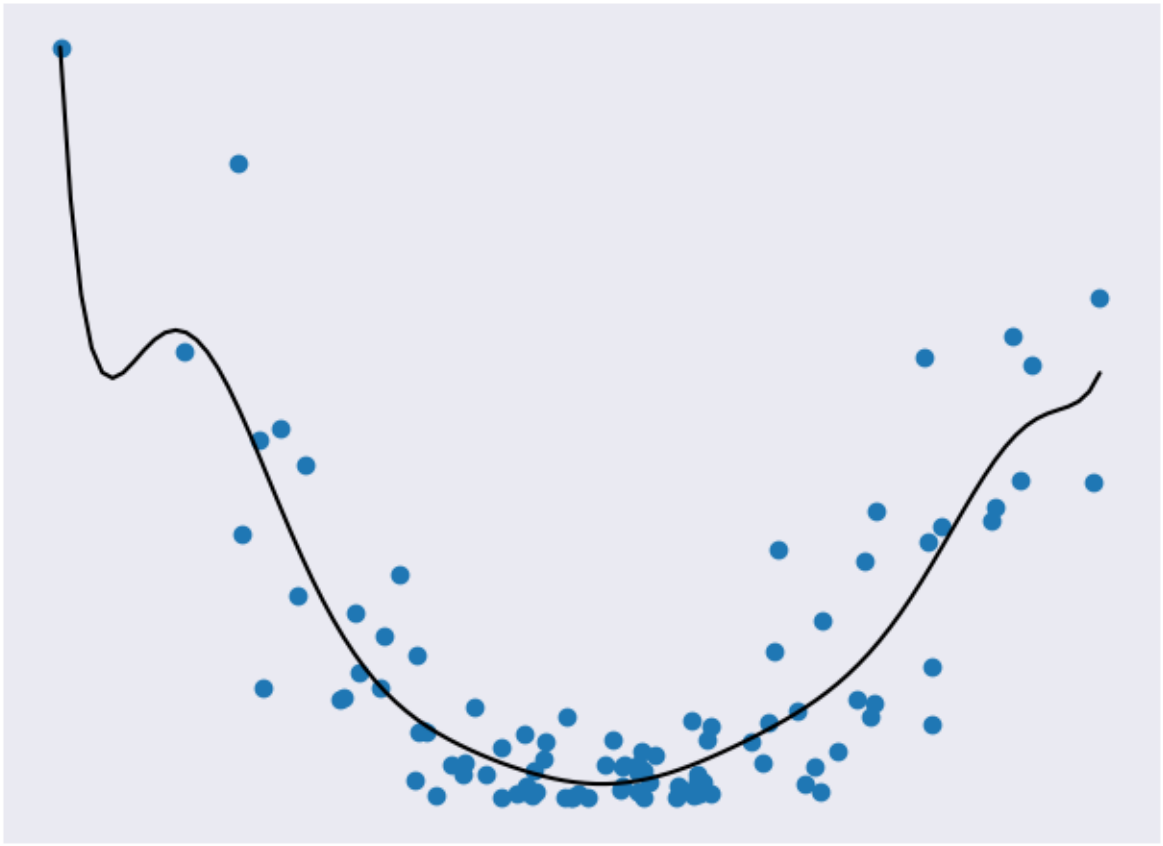


Fig. 3: **Figure 3. A high degree polynomial prediction model** [code]

6.3.1 Ridge Regression

Ridge regression is a type of regularization where the function R involves summing the squares of our weights. *Equation 1* shows an example of the modified cost function.

Fig. 4: **Equation 1. A cost function for ridge regression**

Equation 1 is an example of the regularization with w representing our weights. Ridge regression forces weights to approach zero but will never cause them to be zero. This means that all the features will be represented in our model but overfitting will be minimized. Ridge regression is a good choice when we don't have a very large number of features and just want to avoid overfitting. *Figure 4* gives a comparison of a model with and without ridge regression applied.

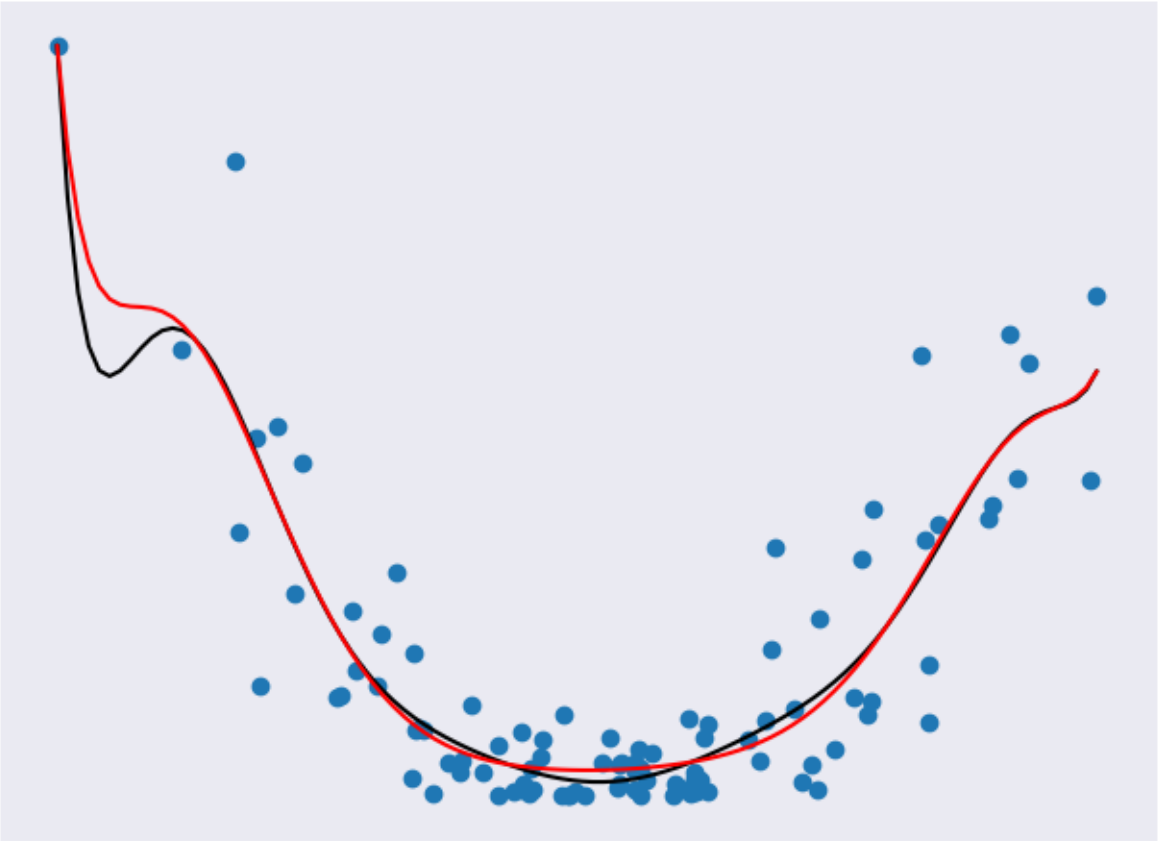


Fig. 5: **Figure 4. Ridge regression applied to a model** [code]

In *Figure 4*, the black line represents a model without Ridge regression applied and the red line represents a model with Ridge regression applied. Note how much smoother the red line is. It will probably do a better job against future data.

In the included regularization_ridge.py file, the code that adds ridge regression is:

Adding the Ridge regression is as simple as adding an additional argument to our Pipeline call. Here, the parameter alpha represents our tuning variable. For additional information on Ridge regression in scikit-learn, consult here.

6.3.2 Lasso Regression

Lasso regression is a type of regularization where the function R involves summing the absolute values of our weights. *Equation 2* shows an example of the modified cost function.

Fig. 6: **Equation 2. A cost function for lasso regression**

Equation 2 is an example of the regularization with w representing our weights. Notice how similar ridge regression and lasso regression are. The only noticeable difference is that square on the weights. This happens to have a big impact on what they do. Unlike ridge regression, lasso regression can force weights to be zero. This means that our resulting model may not even consider some of the features! In the case we have a million features where only a small amount are important, this is an incredibly useful result. Lasso regression lets us avoid overfitting and focus on a small subset of all our features. In the original scenario, we would end up ignoring those factors that don't have as much impact on our sandwich eating experience. *Figure 5* gives a comparison of a model with and without lasso regression applied.

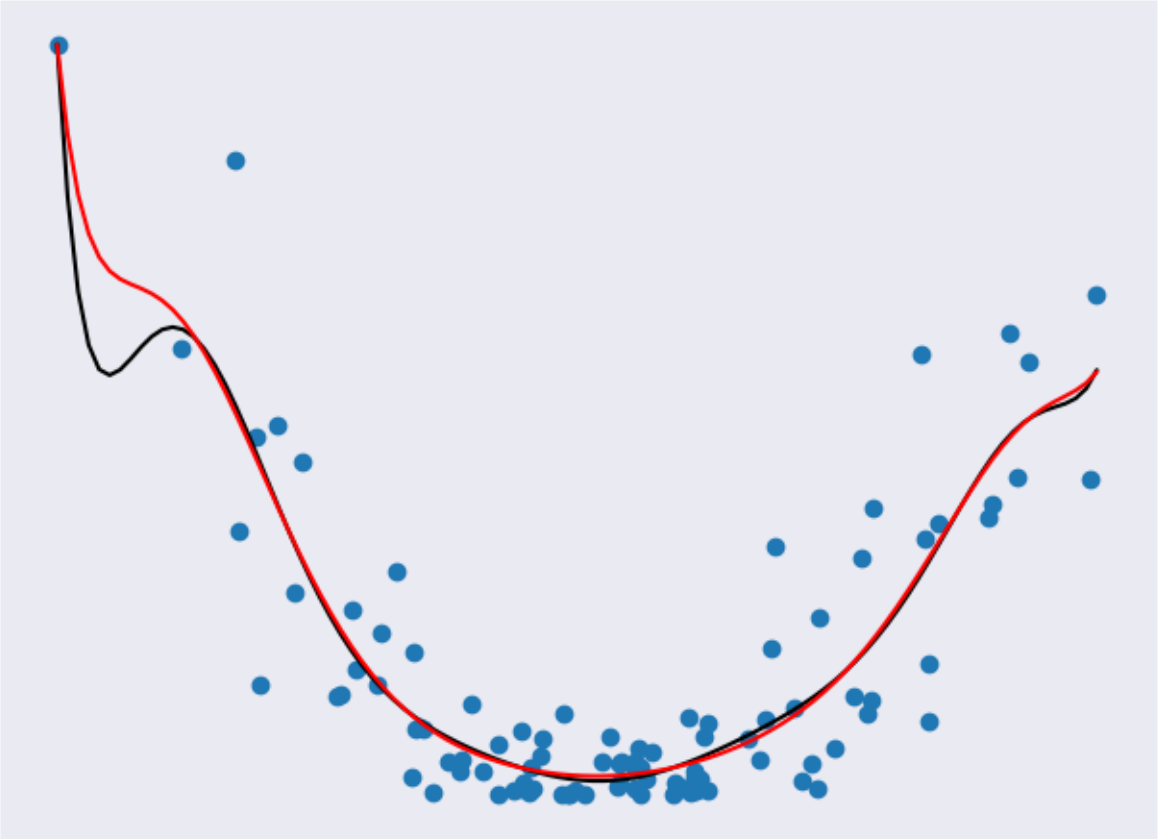


Fig. 7: **Figure 5. Lasso regression applied to a model** [code]

In the figure above, the black line represents a model without Lasso regression applied and the red line represents a model with Lasso regression applied. The red line is much smoother than the black line. The Lasso regression was applied to a model of degree 10 but the result looks like it has a much lower degree! The Lasso model will probably do a better job against future data.

In the included regularization_lasso.py file, the code that adds Lasso regression is:

Adding the Lasso regression is as simple as adding the Ridge regression. Here, the parameter alpha represents our tuning variable and `max_iter` represents the max number of iterations to run for. For additional information on Lasso regression in scikit-learn, consult here.

6.4 Summary

In this module, we learned about regularization. With regularization, we have found a good way to avoid overfitting our data. This is a common but important problem in modeling so it's good to know how to mediate it. We have also explored some methods of regularization that we can use in different situations. With this, we have learned enough about the core concepts of machine learning to move onto our next major topic, supervised learning.

6.5 References

1. https://towardsdatascience.com/regularization-in-machine-learning-76441ddcf99a
2. https://www.analyticsvidhya.com/blog/2018/04/fundamentals-deep-learning-regularization-techniques
3. https://www.quora.com/What-is-regularization-in-machine-learning
4. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html
5. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Lasso.html

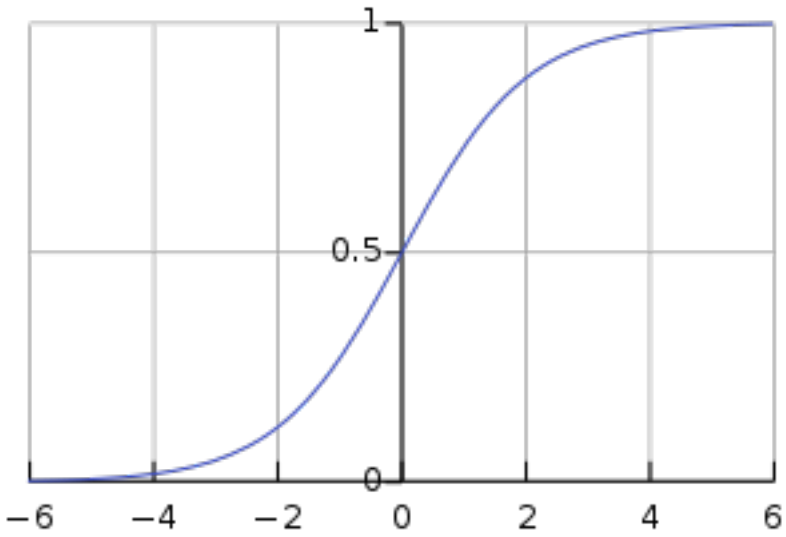
CHAPTER 7

Logistic Regression

7.1 Introduction

Logistic regression is a method for binary classification. It works to divide points in a dataset into two distinct classes, or categories. For simplicity, let's call them class A and class B. The model will give us the probability that a given point belongs in category B. If it is low (lower than 50%), then we classify it in category A. Otherwise, it falls in class B. It's also important to note that logistic regression is better for this purpose than linear regression with a threshold because the threshold would have to be manually set, which is not feasible. Logistic regression will instead create a sort of S-curve (using the sigmoid function) which will also help show certainty, since the output from logistic regression is not just a one or zero. Here is the standard logistic function, note that the output is always between 0 and 1, but never reaches either of those values.

Ref: https://en.wikipedia.org/wiki/Logistic_regression



7.2 When to Use

Logistic regression is great for situations where you need to classify between two categories. Some good examples are accepted and rejected applicants and victory or defeat in a competition. Here is an example table of data that would be a good candidate for logistic regression.

Studying		Success
Hours	Focused	Pass?
1	False	False
3	False	True
0.5	True	False
2	False	True

Notice that the student's success is determined by the inputs and the value is binary, so logistic regression will work well for this scenario.

7.3 How does it work?

Logistic regression works using a linear combination of inputs, so multiple information sources can govern the output of the model. The parameters of the model are the weights of the various features, and represent their relative importance to the result. In the equation that follows, you should recognize the formula used in linear regression. Logistic regression is, at its base, a transformation from a linear predictor to a probability between 0 and 1.

Ref: https://en.wikipedia.org/wiki/Logistic_regression

As in linear regression, the beta values are the weights and x values are the variable inputs. This formula gives the probability that the input belongs to Class B, which is the goal of the logistic regression model.

7.4 Multinomial Logistic Regression

Until now, we've been discussing the situation where there are exactly two distinct outputs, for example a pass or a fail. But, what if there were more than two possible outputs? What about the number classification example, where the output can be any digit from 0 to 9?

Well, there is a way to handle that with logistic regression. When using the scikit-learn library, as the example code does, the facility is already there. With scikit-learn, you can use the multinomial mode and supply any number of classes in the training data. You can think of the method as creating multiple models and comparing their probabilities, but the exact details are beyond the scope of this course.

7.5 Code

Check out the example for logistic regression in our repository.

In the example, scikit-learn and numpy are used to train a simple logistic regression model. The model is basic, but extensible. With logistic regression, more features could be added to the data set seamlessly, simply as a column in the 2D arrays.

The code creates a 2D array representing the training input, in this case it is 1000 x 1, since there are 1000 samples and 1 feature. These inputs are scores out of 1000. A training output array is also created, with the classification of 1 for pass and 0 for fail, based on a threshold. Then, scikit-learn's LogisticRegression class is used to fit a logistic regression classifier to the data. After that, the next step is to test for accuracy with a different data set. So, we create another 100 random samples to test against, and predict against them using the model.

7.6 Motivation

Why use logistic regression? Logistic regression is well suited to the case of **binary classification**, or classifying in 2 categories. Logistic regression is also a relatively simple method, utilizing a weighted sum of inputs, similar to linear regression. Logistic regression is also useful in that it gives a continuous value, representing the probability of a given classification being correct. For these reasons, advocates say that logistic regression should be the first thing learned in the data science world.

7.7 Conclusion

Logistic regression build upon linear regression by extending its use to classification. Although it is not able to classify into more than two classes, it is still effective in what it does, and simple to implement. Consider logistic regression as the first thought pass/fail method. When you just need a pass/fail probability from data, logistic regression is the simplest and likely best option.

Machine learning libraries make using Logistic Regression very simple. Check out the example code in the repository and follow along. The basic idea is to supply the training data as pairs of input and classification, and the model will be built automatically. As always, keep in mind the basics mentioned in the overview section of this repository, as there is no fool-proof method for machine learning.

7.8 References

1. https://towardsdatascience.com/logistic-regression-b0af09cdb8ad

2. https://medium.com/datadriveninvestor/machine-learning-model-logistic-regression-5fa4ffde5773
3. https://github.com/bfortuner/ml-cheatsheet/blob/master/docs/logistic_regression.rst
4. https://machinelearningmastery.com/logistic-regression-tutorial-for-machine-learning/
5. https://towardsdatascience.com/logistic-regression-a-simplified-approach-using-python-c4bc81a87c31
6. https://hackernoon.com/introduction-to-machine-learning-algorithms-logistic-regression-cbdd82d81a36
7. https://en.wikipedia.org/wiki/Logistic_regression
8. https://en.wikipedia.org/wiki/Multinomial_logistic_regression
9. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html
10. https://towardsdatascience.com/5-reasons-logistic-regression-should-be-the-first-thing-you-learn-when-become-a-data-scientist

CHAPTER 8

Naive Bayes Classification

8.1 Motivation

A recurring problem in machine learning is the need to classify input into some preexisting class. Consider the following example.

Say we want to classify a random piece of fruit we found lying around. In this example, we have three existing fruit categories: apple, blueberry, and coconut. Each of these fruits have three features we care about: size, weight, and color. This information is shown in *Figure 1*.

Table 1: **Figure 1. A table of fruit characteristics**

	Apple	Blueberry	Coconut
Size	Moderate	Small	Large
Weight	Moderate	Light	Heavy
Color	Red	Blue	Brown

We observe the piece of fruit we found and determine it has a moderate size, it is heavy, and it is red. We can compare these features against the features of our known classes to guess what type of fruit it is. The unknown fruit is heavy like a coconut but it shares more features with the apple class. The unknown fruit shares 2 of 3 characteristics with the apple class so we guess that it's an apple. We used the fact that the random fruit is moderately sized and red like an apple to make our guess.

This example is a bit silly but it highlights some fundamental points about classification problems. In these types of problems, we are comparing features of an unknown input to features of known classes in our data set. Naive Bayes classification is one way to do this.

8.2 What is it?

Naive Bayes is a classification technique that uses probabilities we already know to determine how to classify input. These probabilities are related to existing classes and what features they have. In the example above, we choose the class that most resembles our input as its classification. This technique is based around using Bayes' Theorem. If you're unfamiliar with what Bayes' Theorem is, don't worry! We will explain it in the next section.

8.3 Bayes' Theorem

Bayes' Theorem [*Equation 1*] is a very useful result that shows up in probability theory and other disciplines.

$$P(A|B) = \frac{P(A)P(B|A)}{P(B)}$$

Fig. 1: **Equation 1. Bayes' Theorem**

With Bayes' Theorem, we can examine conditional probabilities (the probability of an event happening given another event has happened). P(A|B) is the probability that event A will happen given that event B has already happened. We can determine this value using other information we know about events A and B. We need to know P(B|A) (the probability that event B will happen given that event A has already happened), P(B) (the probability event B will happen), and P(A) (the probability event A will happen). We can even apply Bayes' Theorem to machine learning problems!

8.4 Naive Bayes

Naive Bayes classification uses Bayes' Theorem with some additional assumptions. The main thing we will assume is that features are independent. Assuming independence means that the probability of a set of features occurring given a certain class is the same as the product of all the probabilities of each individual feature occurring given that class. In the case of our fruit example above, being red does not affect the probability of being moderately sized so assuming independence between color and size is fine. This is often not the case in real-world problems where features may have complex relationships. This is why "naive" is in the name. If the math seems complicated, don't worry! The code will handle the number crunching for us. Just remember that we are assuming that features are independent of each other to simplify calculations.

In this technique, we take some input and calculate the probability of it happening given that it belongs to one of our classes. We must do this for **each** of our classes. After we have all these probabilities, we just take the one that's the largest as our prediction for what class the input belongs to.

8.5 Algorithms

Below are some common models used for Naive Bayes classification. We have separated them into two general cases based on what type of feature distributions they use: continuous or discrete. Continuous means real-valued (you can have decimal answers) and discrete means a count (you can only have whole number answers). Also provided are the relevant code snippets for each algorithm.

8.5.1 Gaussian Model (Continuous)

Gaussian models assume features follow a normal distribution. As far as you need to know, a normal distribution is just a specific type of probability distribution where values tend to be close to the average. As you can see in *Figure 2*, the plot of a normal distribution has a bell shape. Values are most frequent around the peak of the plot and tend to be rarer the farther away you go. This is another big assumption because many features do not follow a normal distribution. While this is true, assuming a normal distribution makes our calculations a whole lot easier. We use Gaussian models when features are not counts and include decimal values.

The relevant code is available in the gaussian.py file.

In the code, we try and guess a color from given RGB percentages. We create some data to work with where each data point represents an RGB triple. The values of the triples are decimals ranging from 0 to 1 and each has a color class it is associated with. We create a Gaussian model and fit it to the data. We then make a prediction with new input to see which color it should be classified as.

8.5.2 Multinomial Model (Discrete)

Multinomial models are used when we are working with discrete counts. Specifically, we want to use them when we are counting how often a feature occurs. For example, we might want to count how often the word "count" appears on this page. *Figure 3* shows the sort of data we might use with a multinomial model. If we know the counts will only be one of two values, we should use a Bernoulli model instead.

Table 2: **Figure 3. A table of word frequencies for this page**

Word	Frequency
Algebra	0
Big	1
Count	2
Data	12

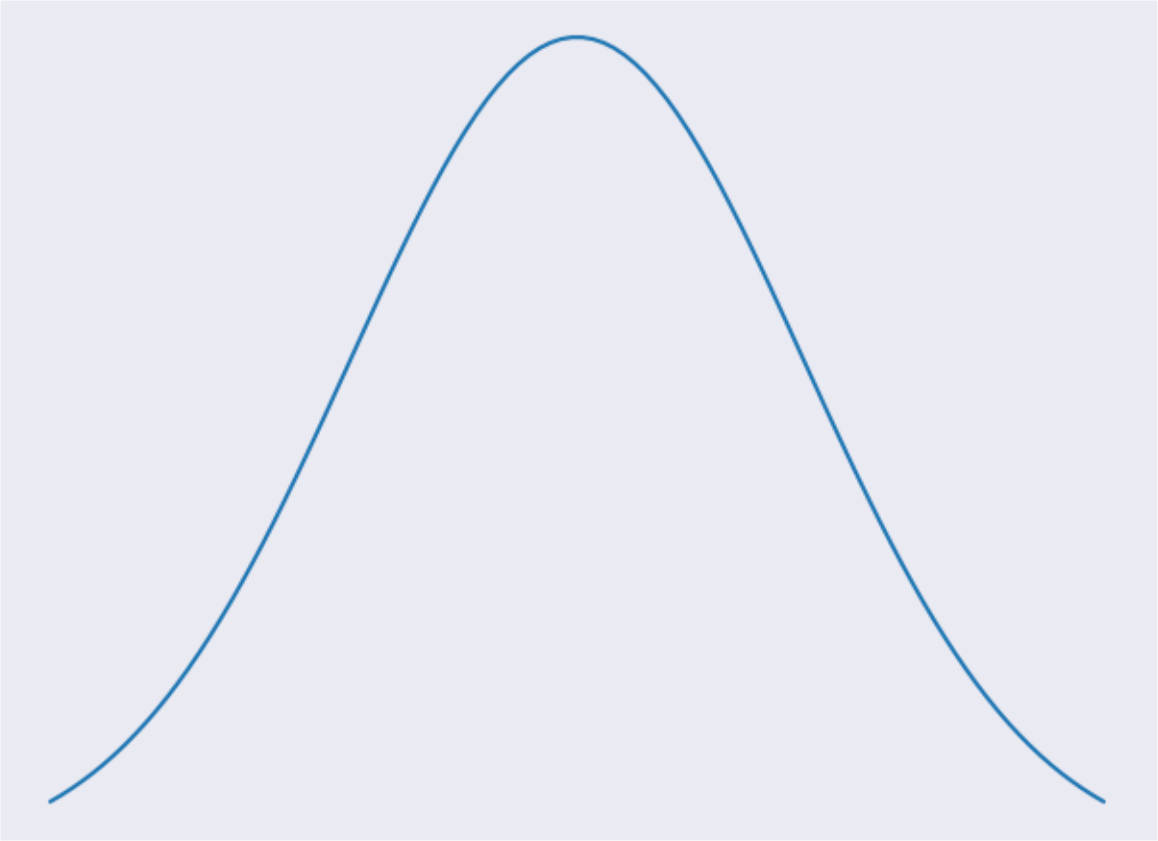


Fig. 2: **Figure 2. A normal distribution with the iconic bell curve shape** [code]

The relevant code is available in the multinomial.py file.

The code is based on our fruit example. In the code, we try and guess a fruit from given characteristics. We create some data to work with where each data point is a triple representing characteristics of a fruit namely size, weight, and color. The values of the triples are integers ranging from 0 to 2 and each has a fruit class it is associated with. The integers are basically just labels associated with characteristics but using them instead of strings allows us to use a Multinomial model. We create a Multinomial model and fit it to the data. We then make a prediction with new input to see which fruit it should be classified as.

8.5.3 Bernoulli Model (Discrete)

Bernoulli models are also used when we are working with discrete counts. Unlike the multinomial case, here we are counting whether or not a feature occurred. For example, we might want to check if the word "count" appears at all on this page. We can also use Bernoulli models when features only have 2 possible values like red or blue. *Figure 4* shows the sort of data we might use with a Bernoulli model.

Table 3: **Figure 4. A table of word appearances on this page**

Word	Present?
Algebra	False
Big	True
Count	True
Data	True

The relevant code is available in the bernoulli.py file.

In the code, we try and guess if something is a duck or not based on certain characteristics it has. We create some data to work with where each data point is a triple representing the characteristics: walks like a duck, talks like a duck, and is small. The values of the triples are either 1 or 0 for true or false and each is either a duck or not a duck. We create a Bernoulli model and fit it to the data. We then make a prediction with new input to see whether or not it is a duck.

8.6 Conclusion

In this module, we learned about Naive Bayes classification. Naive Bayes classification lets us classify an input based on probabilities of existing classes and features. As demonstrated in the code, you don't need a lot of training data for Naive Bayes to be useful. Another bonus is speed which can come in handy for real-time predictions. We make a lot of assumptions to use Naive Bayes so results should be taken with a grain of salt. But if you don't have much data and need fast results, Naive Bayes is a good choice for classification problems.

8.7 References

1. https://machinelearningmastery.com/naive-bayes-classifier-scratch-python/
2. https://www.analyticsvidhya.com/blog/2017/09/naive-bayes-explained/
3. https://towardsdatascience.com/naive-bayes-in-machine-learning-f49cc8f831b4
4. https://medium.com/machine-learning-101/chapter-1-supervised-learning-and-naive-bayes-classification-part-1-theory-8b9e361

CHAPTER 9

Decision Trees

9.1 Introduction

Decision trees are a classifier in machine learning that allows us to make predictions based on previous data. They are like a series of sequential "if . . . then" statements you feed new data into to get a result.

To demonstrate decision trees, let's take a look at an example. Imagine we want to predict whether Mike is going to go grocery shopping on any given day. We can look at previous factors that led Mike to go to the store:

Here we can see the amount of grocery supplies Mike had, the weather, and whether Mike worked each day. Green rows are days he went to the store, and red days are those he didn't. The goal of a decision tree is to try to understand *why* Mike goes to the store, and apply that to new data later on.

Let's divide the first attribute up into a tree. Mike can either have a low, medium, or high amount of supplies:

Here we can see that Mike never goes to the store if he has a high amount of supplies. This is called a **pure subset**, a subset with only positive or only negative examples. With decision trees, there is no need to break a pure subset down

	Supplies	Weather	Worked		Shopped
D1	Low	Sunny	Yes		Yes
D2	High	Sunny	Yes		No
D3	Med	Cloudy	Yes		No
D4	Low	Raining	Yes		No
D5	Low	Cloudy	No		Yes
D6	High	Sunny	No		No
D7	High	Raining	No		No
D8	Med	Cloudy	Yes		No
D9	Low	Raining	Yes		No
D10	Low	Raining	No		Yes
D11	Med	Sunny	No		Yes
D12	High	Sunny	Yes		No

Fig. 1: **Figure 1. An example dataset**

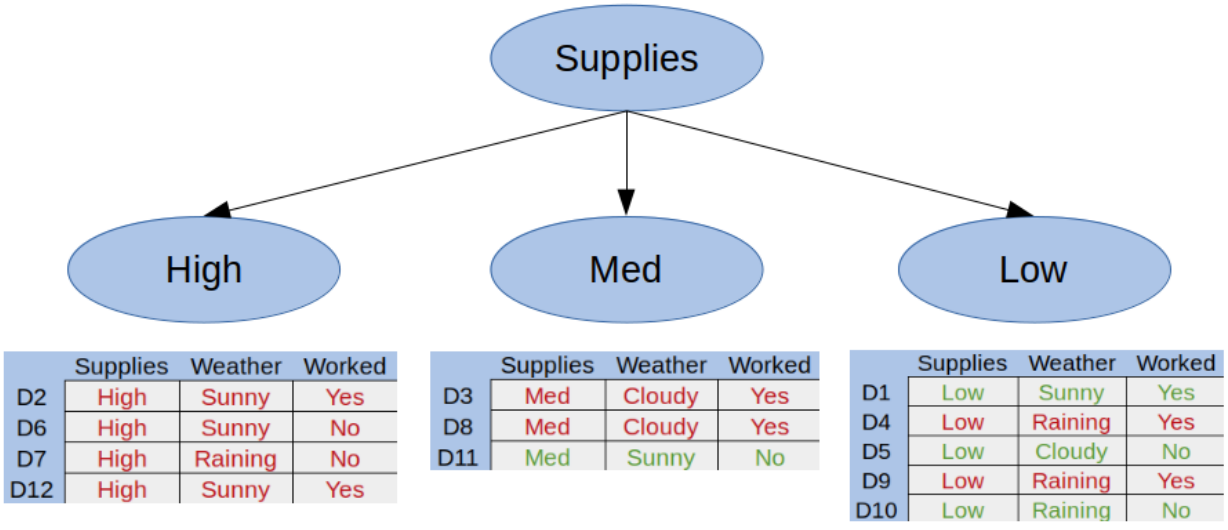


Fig. 2: **Figure 2. Our first split**

further.

Let's break the Med Supplies category into whether Mike worked that day:

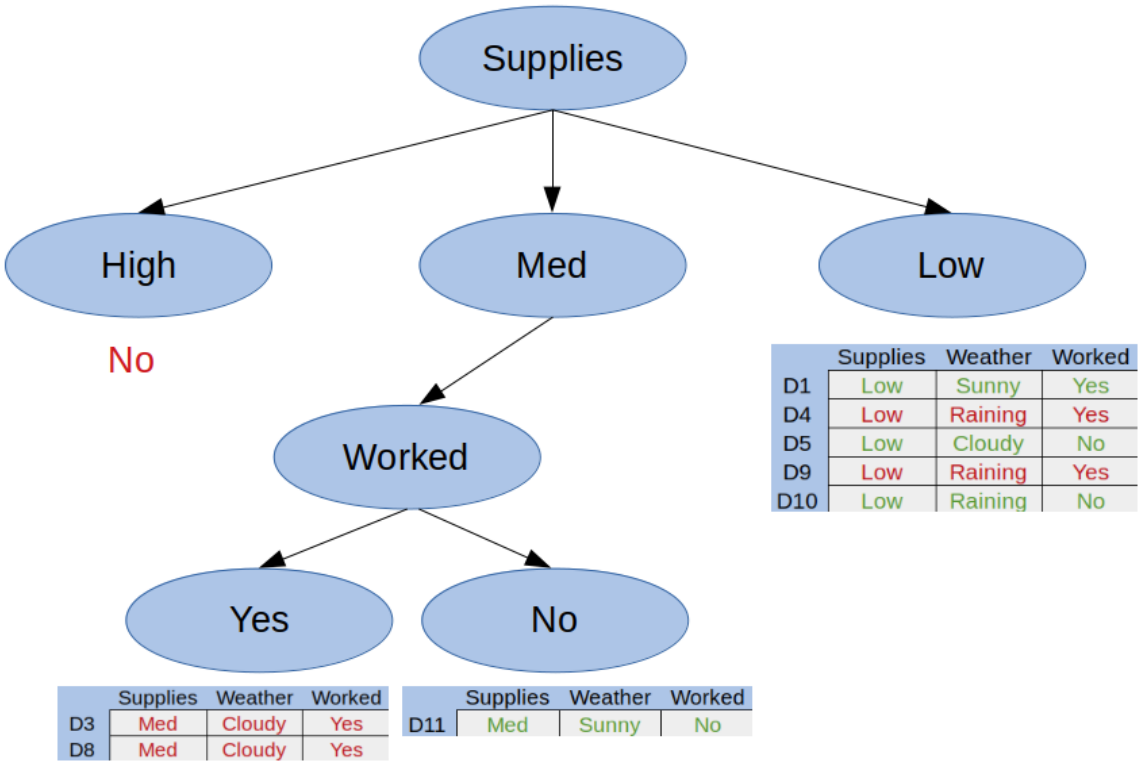


Fig. 3: **Figure 3. Our second split**

Here we can see we have two more pure subsets, so this tree is complete. We can replace any pure subsets with their respective answer - in this case, yes or no.

Finally, let's split the Low Supplies category by the Weather attribute:

Now that we have all pure subsets, we can create our final decision tree:

9.2 Motivation

Decision trees are easily created, visualized, and interpreted. Because of this, they are typically the first method used to model a dataset. The hierarchical structure and categorical nature of a decision tree makes it highly intuitive to implement. Decision trees expand logarithmically based on the number of data points you have, meaning larger datasets will impact the tree creation process less than other classifiers. Because of the tree structure, classifying new data points is also performed logarithmically.

9.3 Classification and Regression Trees

Decision tree algorithms are also known as CART, or Classification and Regression Trees. A **Classification Tree**, like the one shown above, is used to get a result from a set of possible values. A **Regression Tree** is a decision tree where the result is a continuous value, such as the price of a car.

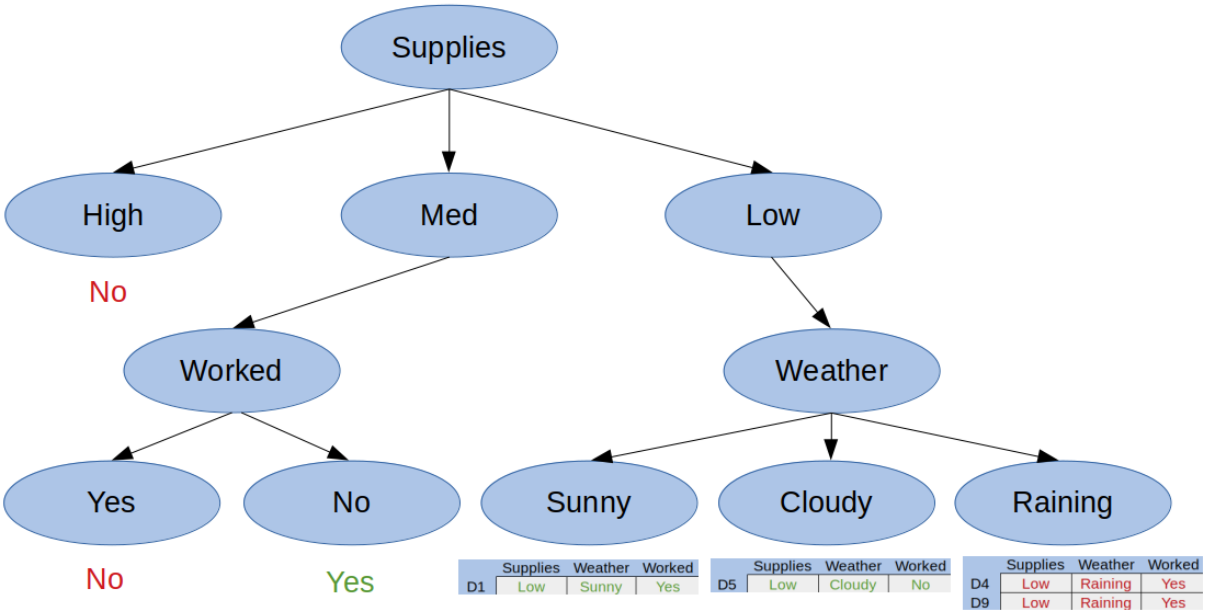


Fig. 4: **Figure 4. Our third split**

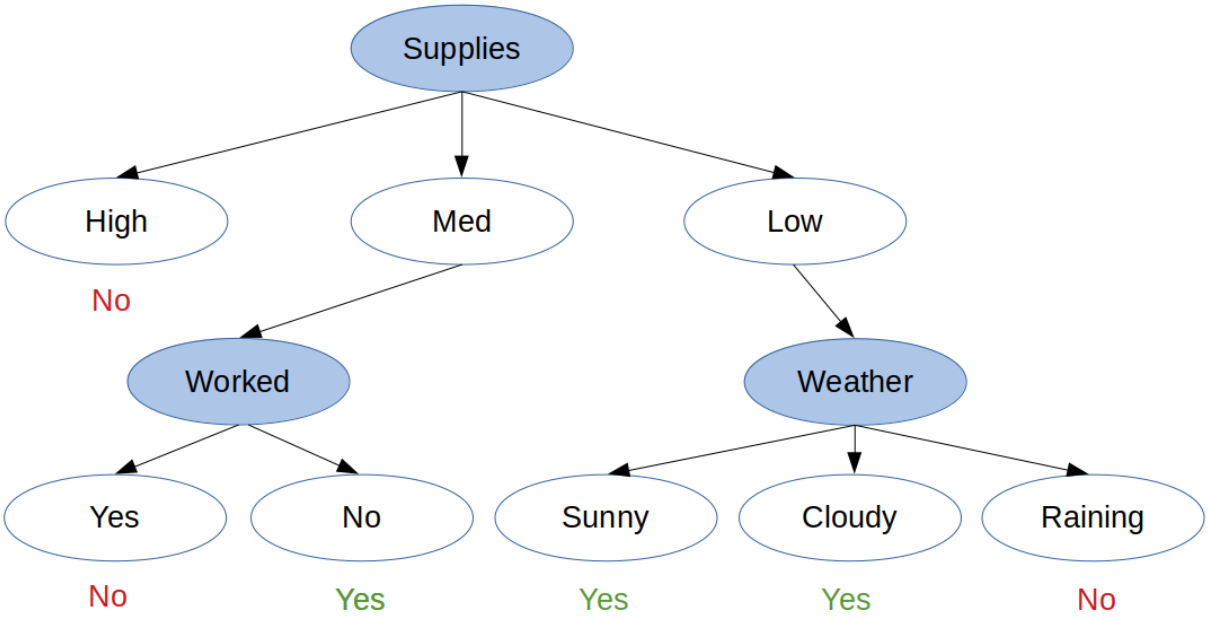


Fig. 5: **Figure 5. The final decision tree**

9.4 Splitting (Induction)

Decision trees are created through a process of splitting called **induction**, but how do we know when to split? We need a recursive algorithm that determines the best attributes to split on. One such algorithm is the **greedy algorithm**:

1. Starting from the root, we create a split for each attribute.
2. For each created split, calculate the cost of the split.
3. Choose the split that costs the least.
4. Recurse into the sub-trees and continue from step 1.

This process is repeated until all nodes have the same value as the target result, or splitting adds no value to a prediction. This algorithm has the root node as the best classifier.

9.5 Cost of Splitting

The cost of a split is determined by a **cost function**. The goal of using a cost function is to split the data in a way that can be computed and that provides the most information gain.

For classification trees, those that provide an answer rather than a value, we can compute imformation gain using *Gini Impurities*:

$$I_G(t) = \sum_{i=1}^{c} p(i|t)\big(1 - p(i|t)\big) = 1 - \sum_{i=1}^{c} p(i|t)^2.$$

Fig. 6: **Equation 1. The Gini Impurity Function**
Ref: https://sebastianraschka.com/faq/docs/decision-tree-binary.html

$$IG(D_p, f) = I(D_p) - \sum_{j=1}^{m} \frac{N_j}{N} I(D_j),$$

Fig. 7: **Equation 2. The Gini Information Gain Formula**
Ref: https://sebastianraschka.com/faq/docs/decision-tree-binary.html

To calculate information gain, we first start by computing the Gini Impurity of our root node. Let's take a look at the data we used earlier:

	Supplies	Weather	Worked?	Shopped?
D1	Low	Sunny	Yes	Yes
D2	High	Sunny	Yes	No
D3	Med	Cloudy	Yes	No
D4	Low	Raining	Yes	No
D5	Low	Cloudy	No	Yes
D6	High	Sunny	No	No
D7	High	Raining	No	No
D8	Med	Cloudy	Yes	No
D9	Low	Raining	Yes	No
D10	Low	Raining	No	Yes
D11	Med	Sunny	No	Yes
D12	High	Sunny	Yes	No

Our root node is the target variable, whether Mike is going to go shopping. To calculate its Gini Impurity, we need to find the sum of probabilities squared for each outcome and subtract this result from one:

$$I_G(Shopped) = 1 - (P(”yes”)^2 + P(”no”)^2)$$

$$I_G(Shopped) = 1 - ((\frac{4}{12})^2 + (\frac{8}{12})^2)$$

$$I_G(Shopped) = 0.\overline{44}$$

Let’s calculate the Gini Information Gain if we split on the first attribute, Supplies. We have three different categories we can split by - Low, Med, and High. For each of these, we calculate its Gini Impurity:

As you can see, the impurity for High supplies is 0. This means that if we split on Supplies and receive High input, we immediately know what the outcome will be. To determine the Gini Information Gain for this split, we compute the root’s impurity minus the weighted average of each child’s impurity:

We continue this pattern for every possible split, then choose the split that gives us the highest information gain value. Maximizing information gain leaves us with the most polarized splits possible, lowering the probability new input is incorrectly classified.

9.6 Pruning

A decision tree created through a sufficiently large dataset may end up with an excessive amount of splits, each with decreasing usefulness. A highly detailed decision tree can even lead to overfitting, discussed in the previous module. Because of this, it’s beneficial to prune less important splits of a decision tree away. Pruning involves calculating the information gain of each ending sub-tree (the leaf nodes and their parent node), then removing the sub-tree with the least information gain:

As you can see, the sub-tree is replaced with the more prominent result, becoming a new leaf. This process can be repeated until you reach a desired complexity level, tree height, or information gain amount. Information gain can be tracked and stored as the tree is built to save time when pruning as well. Each model should make use of its own pruning algorithm to meet its needs.

$$I_G(Low) = 1 - ((\frac{3}{5})^2 + (\frac{2}{5})^2) = 0.48$$

$$I_G(Med) = 1 - ((\frac{1}{3})^2 + (\frac{2}{3})^2) = 0.\overline{44}$$

$$I_G(High) = 1 - ((\frac{0}{4})^2 + (\frac{4}{4})^2) = 0$$

$$IG_G = I_G(Supplies) - (\frac{N_{Low}}{N} I_G(Low) + \frac{N_{Med}}{N} I_G(Med) + \frac{N_{High}}{N} I_G(High))$$

$$IG_G = 0.\overline{44} - (\frac{5}{12}(0.48) + \frac{3}{12}(0.\overline{44}) + \frac{4}{12}(0)) = 0.1\overline{33}$$

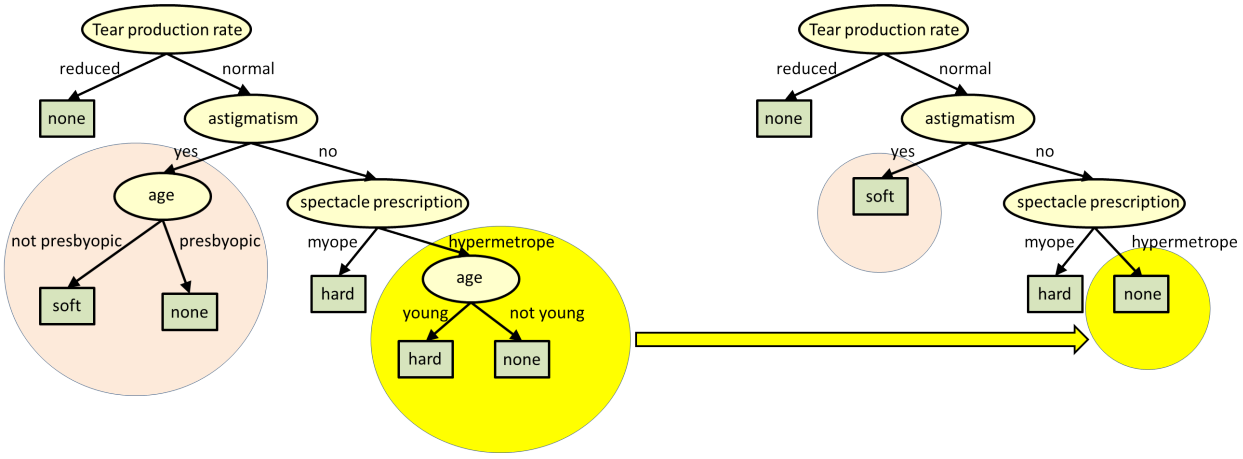


Fig. 8: Ref: http://www.cs.cmu.edu/~bhiksha/courses/10-601/decisiontrees/

9.7 Conclusion

Decision trees allow you to quickly and efficiently classify data. Because they shape data into a heirarchy of decisions, they are highly understandable by even non-experts. Decision trees are created and refined in a two-step process - induction and pruning. Induction involves picking the best attribute to split on, while pruning helps to filter out results deemed useless. Because decision trees are so simple to create and understand, they are typically the first approach used to model and predict outcomes of a dataset.

9.8 Code Example

The provided code, decisiontrees.py takes the example discussed in this documentation and creates a decision tree from it. First, each possible option for each class is defined. This is used later to fit and display our decision tree:

```
# The possible values for each class
classes = {
    'supplies': ['low', 'med', 'high'],
    'weather':  ['raining', 'cloudy', 'sunny'],
    'worked?':  ['yes', 'no']
}
```

Next, we've created a matrix of the dataset shown above and defined each row's outcome:

```
# Our example data from the documentation
data = [
    ['low',  'sunny',   'yes'],
    ['high', 'sunny',   'yes'],
    ['med',  'cloudy',  'yes'],
    ['low',  'raining', 'yes'],
    ['low',  'cloudy',  'no' ],
    ['high', 'sunny',   'no' ],
    ['high', 'raining', 'no' ],
    ['med',  'cloudy',  'yes'],
    ['low',  'raining', 'yes'],
    ['low',  'raining', 'no' ],
    ['med',  'sunny',   'no' ],
    ['high', 'sunny',   'yes']
]

# Our target variable, whether someone went shopping
target = ['yes', 'no', 'no', 'no', 'yes', 'no', 'no', 'no', 'no', 'yes', 'yes', 'no']
```

Unfortunately, the sklearn machine learning package can't create a decision tree from categorical data. There is in-progress work to allow this, but for now we need another way to represent the data in a decision tree with the library. A naive approach would be to just enumerate each category - for instance, converting sunny/raining/cloudy to values such as 0, 1, and 2. There are some unfortunate side effects of doing this though, such as the values being comparable (sunny < raining) and continuous. To get around this, we "one hot encode" the data:

```
categories = [classes['supplies'], classes['weather'], classes['worked?']]
encoder = OneHotEncoder(categories=categories)

x_data = encoder.fit_transform(data)
```

One hot encoding allows us to convert categorical data into values recognizable by ML algorithms expecting continuous data. It works by taking a class and dividing it up into each option, with a bit representing whether the option is present.

Now that we have data suited to sklearn's decision tree model, we simply fit the classifier to the data:

```
# Form and fit our decision tree to the now-encoded data
classifier = DecisionTreeClassifier()
tree = classifier.fit(x_data, target)
```

The rest of the code involves creating some random prediction input to show how you can use the tree. We create a random set of data in the same format as the data above, then pass it into DecisionTreeClassifier's predict method. This gives us an array of predicted target variables - in this case, yes or no answers to whether Mike will go shopping:

```
# Use our tree to predict the outcome of the random values
prediction_results = tree.predict(encoder.transform(prediction_data))
```

9.9 References

1. https://towardsdatascience.com/decision-trees-in-machine-learning-641b9c4e8052
2. https://heartbeat.fritz.ai/introduction-to-decision-tree-learning-cd604f85e23
3. https://machinelearningmastery.com/implement-decision-tree-algorithm-scratch-python/
4. https://sebastianraschka.com/faq/docs/decision-tree-binary.html
5. https://www.cs.cmu.edu/~bhiksha/courses/10-601/decisiontrees/

CHAPTER 10

k-Nearest Neighbors

10.1 Introduction

K-Nearest Neighbors (KNN) is a basic classifier for machine learning. A **classifier** takes an already labeled data set, and then it trys to label new data points into one of the catagories. So, we are trying to identify what class an object is in. To do this we look at the closest points (neighbors) to the object and the class with the majority of neighbors will be the class that we identify the object to be in. The k is the number of nearest neighbors to the object. So, if k = 1 then the class the object would be in is the class of the closest neighbor. Let's look at an example.

In this example we are trying to classify the red star to be either a green square or a blue octagon. First, if we look at the inner circle where k = 3, we can see that there are 2 blue octagons and 1 green square. So there is a majority of blue octagons, so the red star would be classified as a blue octagon. Now we look at k = 5, the outer circle. In this one there is 2 blue octagons and 3 green squares. Then, the red star would be classified as a green square.

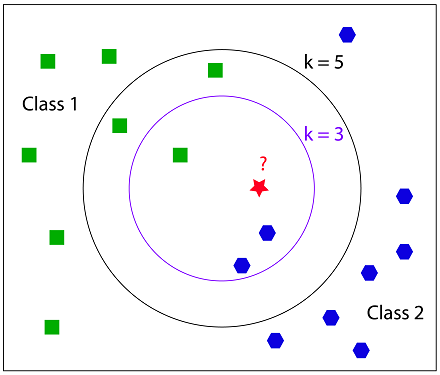


Fig. 1: Ref: https://coxdocs.org

10.2 How does it work?

We will look at two different ways to go about this. The two ways are the brute force method and the K-D tree method.

10.3 Brute Force Method

This is the simplest method. Basically, it's just calculating the **Euclidean distance** from the object being classified to each point in the set. The Euclidean distance is simply the length of a line segment that connects two points. The Brute Force method is useful when the dimensions of the points are small or the number of points is small. As the number of points increases the number of times the method will have to calculate the Euclidean distance also increases, so the performance of the method drops. Luckily, the K-D tree method is better equipped for larger sets of data.

10.4 K-D Tree Method

This method tries to improve the running time by reducing the amount of times we calculate the Euclidean distance. The idea behind this method is that if we know that two data points are close to each other and we calculate the Euclidean distance to one of them and then we know that distance is roughly close to the other point. Here is an example of how the K-D tree looks like.

How a K-D tree works is that a node in the tree represents and holds data from an n-dimensional graph. Each node represents a box in the graph. First we can build a K-D tree out of a set of data, then when it's time to classify a point we would just look at where the point will fall in the tree then calculate the Euclidean distance between only the points it is close to until we reach k neighbors.

If you have a larger data set it is recommended to use this method. This is because the cost of creating the K-D tree is relatively low if the data set is larger, and the cost of classifiying a point is constant as the data gets larger.

10.5 Choosing k

Choosing k typically depends on the dataset you are looking at. You never want to choose k = 2 because it has a very high chance that there won't be a majority class, so in the example above there would be one of each so we wouldn't be able to classify the red star. Typically, you want the value of k to be small. As k goes to infinity all unidentified data

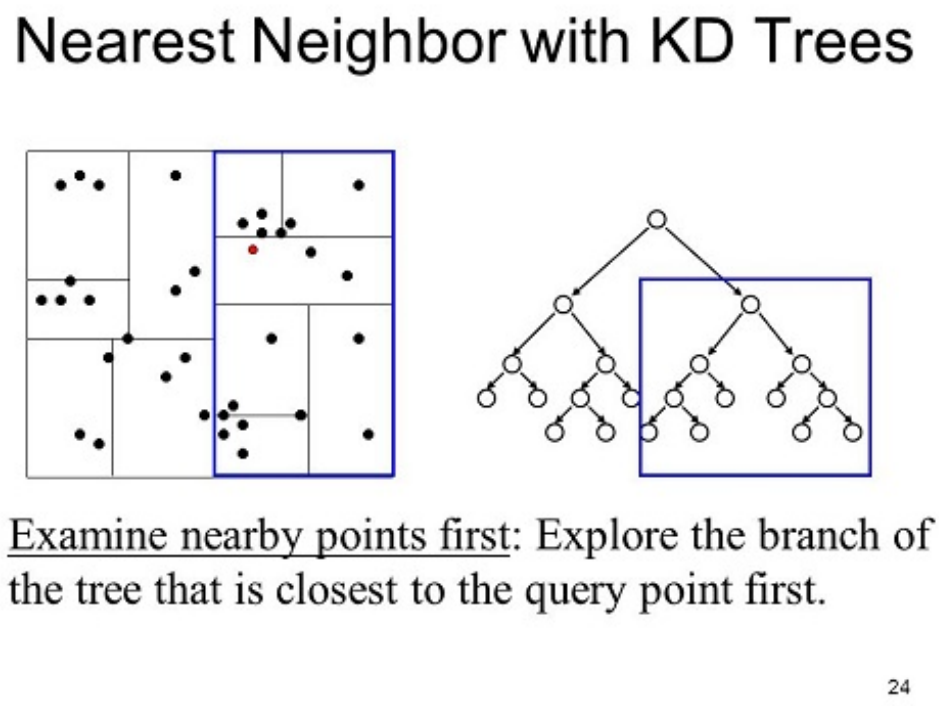


Fig. 2: Ref: https://slideplayer.com/slide/3273367/

points will always be classified to one class or the other depending on which class has more data points. You don't want this to happen, so it is wise to choose a k that is relatively small.

10.6 Conclusion

Here are some things to take away:

- The different methods to KNN only affect the performance, not the output
- The Brute Force Method is best when the dimensions of the points or the number of points are small
- The K-D Tree Method is best when you have a larger data set
- SKLearn KNN classifier has a auto method which decides what method to use given what data it's trained on.

Choosing the value of k will drastically change how the data is classified. A higher k value will ignore outliers to the data and a lower will give more weight to them. If the k value is too high it will not be able to classify the data, so k needs to be relatively small.

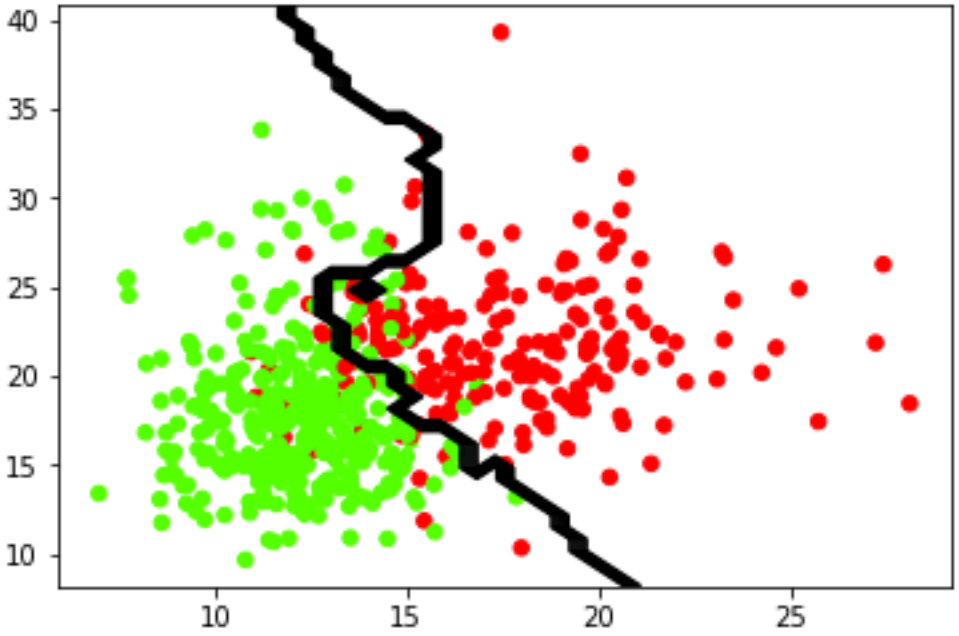
10.7 Motivation

So why would someone use this classifier over another? Is this the best classifier? The answer to these questions are that it depends. There is no classifier that is best, it all depends on the data that a classifier is given. KNN might be the best for one dataset but not another. It's good to know about other classifiers like Support Vector Machines, and then decide which one best classifies the a given dataset.

10.8 Code Example

Check out our code, knn.py to learn how to implement a k nearest neighbor classifier using Python's Scikit-learn library. More information about Scikit-Learn can be found here.

knn.py, Classifies a set of data on breast cancer, loaded from Scikit-Learn's dataset library. The program will take the data and plot them on a graph, then use the KNN algorithm to best separate the data. The output should look like this:



The green points are classified as benign. The red points are classified as malignant. The boundary line is the prediction that the classifier makes. This boundary line is determined by the k value, for this instance k = 9.

This loads the data from the Scikit-Learn's dataset library. You can change the data to whatever you would like. Just make sure you have data points and an array of targets to classify those data points.

```
dataCancer = load_breast_cancer()
data = dataCancer.data[:, :2]
target = dataCancer.target
```

You can also change the k value or n_neighbors value that will change the algorithm. It is suggested that you choose a k that is relatively small.

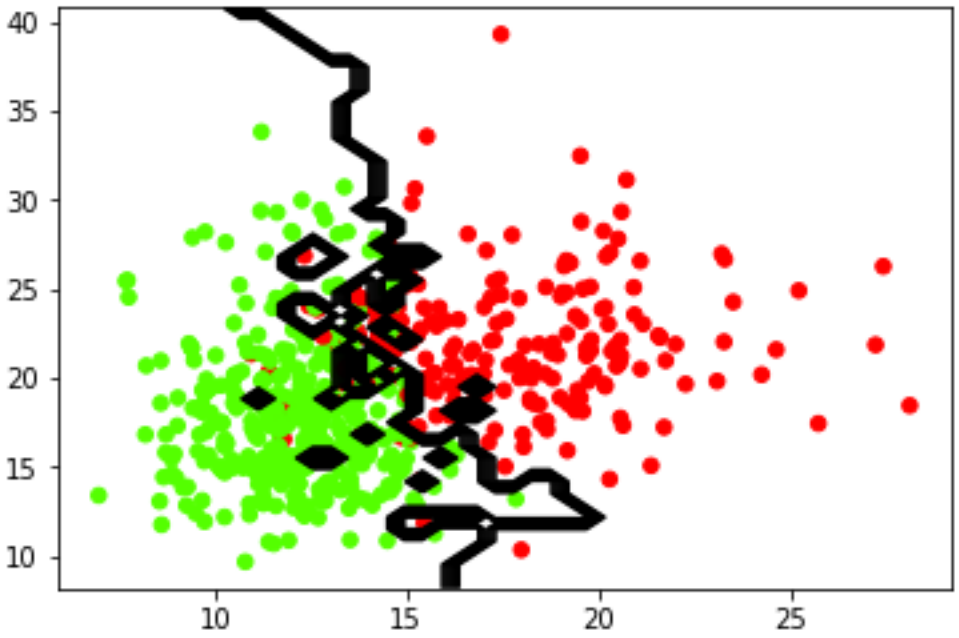
You can also change the algorithm used, the options are {'auto', 'ball_tree', 'kd_tree', 'brute'}. These don't change the output of the prediction, they will just change the time it takes to predict the data.

Try changing the value of n_neighbors to 1 in the code below.

```
model = KNeighborsClassifier(n_neighbors = 9, algorithm = 'auto')
model.fit(data, target)
```

If you changed the value of n_neighbors to 1 this will classify by the point that is closest to the point. The output should look like this:

Comparing this output to k = 9 you can see a large difference on how it classifies the data. So if you want to ignore outliers you will want a higher k value, otherwise choose a smaller k like 1, 3 or 5. You can experiment by choosing a



very high k greater than 100. Eventually the algorithm will classify all the data into 1 class, and there will be no line to split the data.

10.9 References

1. https://medium.com/machine-learning-101/k-nearest-neighbors-classifier-1c1ff404d265
2. https://www.analyticsvidhya.com/blog/2018/03/introduction-k-neighbours-algorithm-clustering/
3. https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html
4. https://turi.com/learn/userguide/supervised-learning/knn_classifier.html

CHAPTER 11

Linear Support Vector Machines

11.1 Introduction

A **Support Vector Machine** (SVM for short) is another machine learning algorithm that is used to classify data. The point of SVM's are to try and find a line or **hyperplane** to divide a dimensional space which best classifies the data points. If we were trying to divide two classes A and B, we would try to best separate the two classes with a line. On one side of the line/hyperplane would be data from class A and on the other side would be from class B. This alogorithm is very useful in classifying because we must to calculate the best line or hyperplane once and any new data points can easily be classified just by seeing which side of the line they fall on. This contrasts with the k-nearest neighbors algortihm, where we would have to calculate each data points nearest neighbors.

11.2 Hyperplane

A **hyperplane** depends on the space it is in, but it divides the space into two disconnected parts. For example, 1-dimensional space would just be a point, 2-d space a line, 3-d space a plane, and so on.

11.3 How do we find the best hyperplane/line?

You might be wondering that there could be multiple lines that split the data well. In fact, there is an infinite amount of lines that can divide two classes. As you can see in the graph below, every line splits the squares and the circles, so which one do we choose?

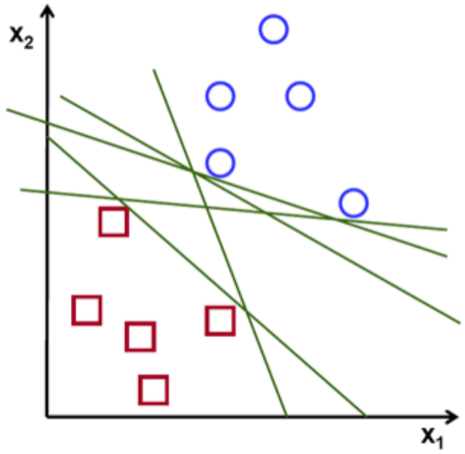


Fig. 1: Ref: https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47

So how does SVM find the ideal line to separate the two classes? It doesn't just pick a random one. The algorithm chooses the line/hyperplane with the **maximum margin**. Maximizing the margin will give us the optimal line to classify the data. This is shown in the figure below.

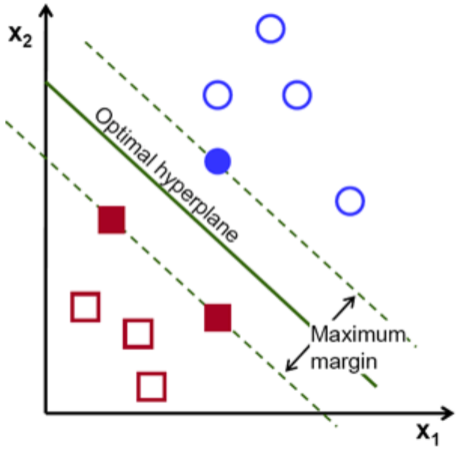


Fig. 2: Ref: https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47

11.4 How to maximize the margin?

The data that is closest to the line is what determines the optimal line. These data points are called **support vectors**. They are shown as the filled in squares and circles above. The distance from these vectors to the hyperplane is called the **margin**. In general, the further those points are from the hyperplane, the greater the probability of correctly classifying the data. There is a lot of complex math that goes into finding the support vectors and maximizing the margin. We won't go into that; we just want to get the basic idea behind SVMs.

11.5 Ignore Outliers

Sometimes data classes will have **outliers**. These are data points that are clearly separated from the rest of their class. Support Vector Machines will ignore these outliers. This is shown in the figure below.

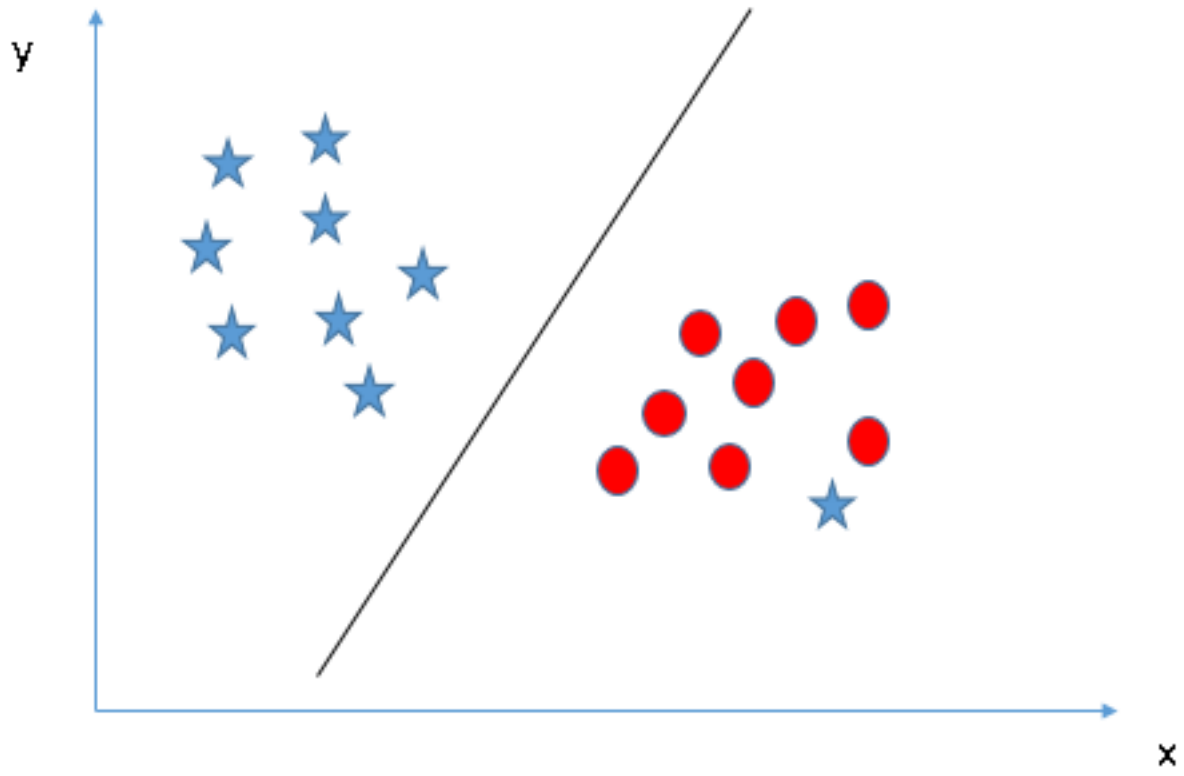


Fig. 3: Ref: https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code/

The star that is with the red circles is the outlier. So, the SVM ignores the outlier and creates the best line to separate the two classes.

11.6 Kernel SVM

There will be data classes that can't be separated with a simple line or hyperplane. This is called **non-linearly separable data**. Here is an example of that kind of data.

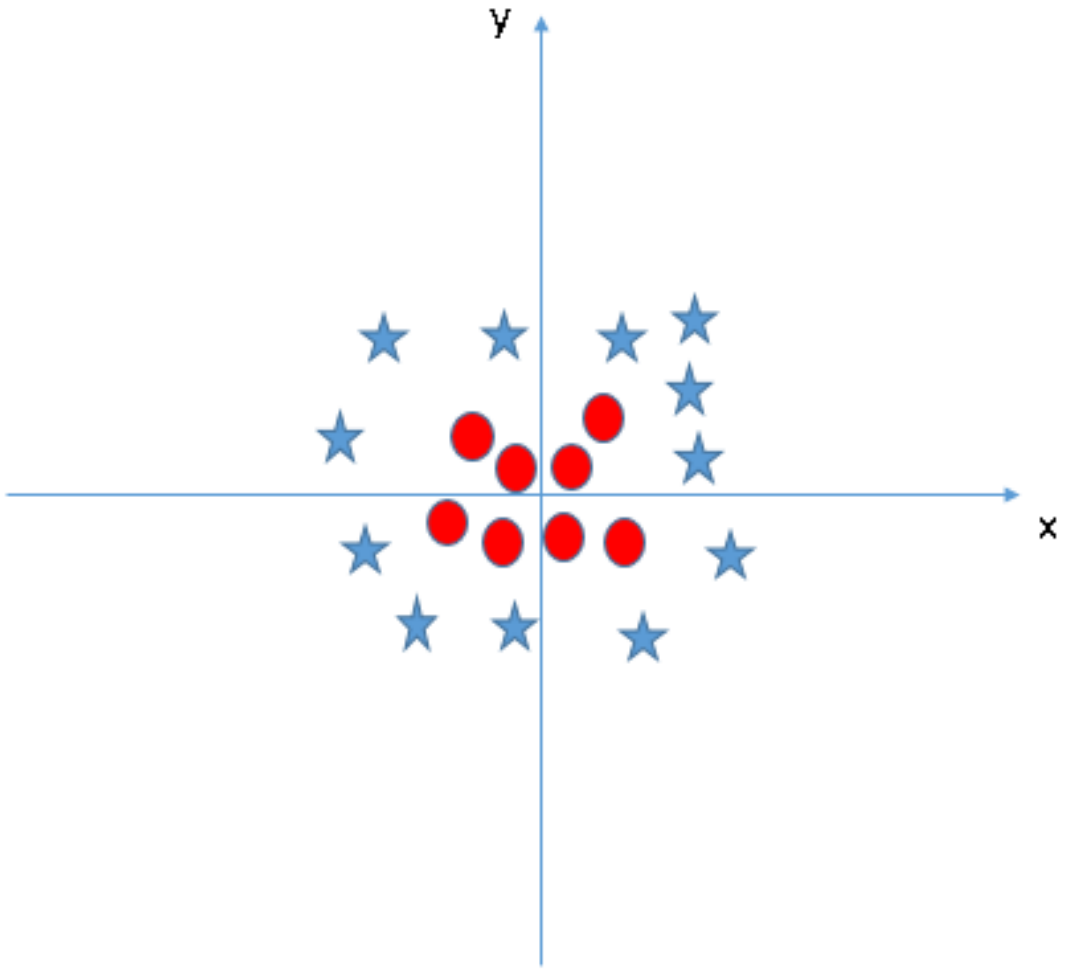


Fig. 4: Ref: https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code/

There is no clear way to separate the stars from the circles. SVMs will be able to classify non-linearly separable data by using a trick called the **kernel trick**. Basically, the kernel trick takes the points to a higher dimension to turn non-linearly separable data to linear separable data. So the above figure would be classified with a circle that separates the data.

Here is an example of the kernel trick.

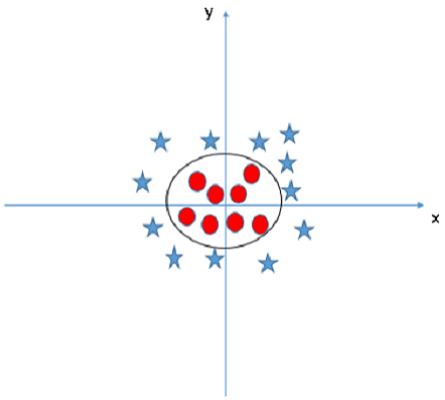


Fig. 5: Ref: https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code/

There are three types of kernels:

- **Linear** Kernel
- **Polynomial** Kernel
- **Radial Basis Function (RBF)** kernel

You can see how these kernels change the outcome of the optimal hyperplane by changing the value of kernel in "model = svm.SVC(kernel = 'linear', C = 10000)" to either 'poly' or 'rbf'. This is in the linear_svm.py.

11.7 Conclusion

An SVM is a great machine learning technique to classify data. Now that we know a little about SVM's we can show the advantages and disadvantages to using this classifier.

The pros to SVM's:

- Effective in classifying higher dimensional space
- Saves space on memory because it only uses the support vectors to create the optimal line.
- Best classifier when data points are separable

The cons to SVM's:

- Performs poorly when there is a large data set, the training times are longer.
- Performs badly when the classes are overlapping, i.e. non-separable data points.

11.8 Motivation

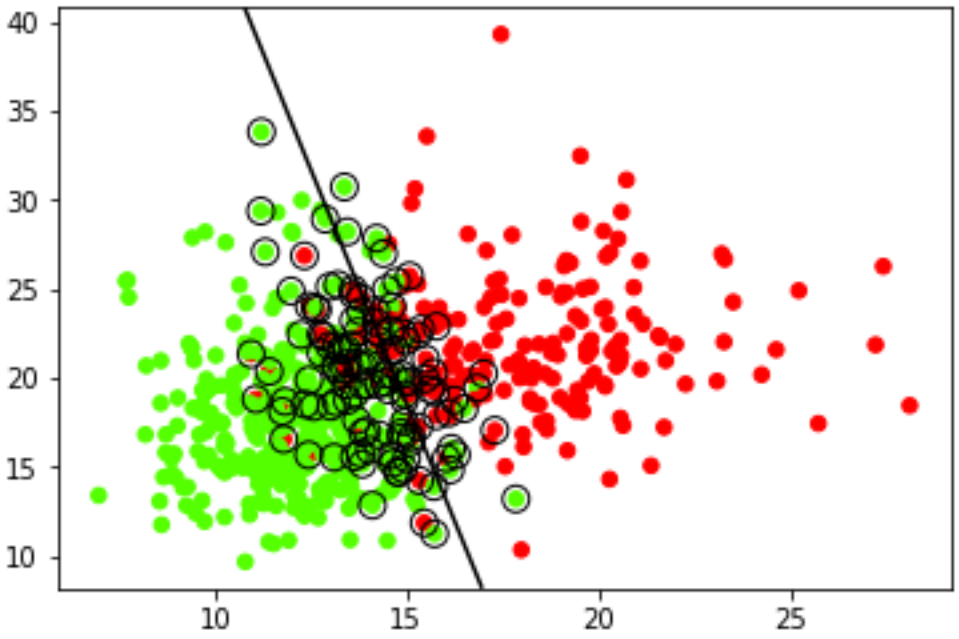
Why would you ever use SVMs? There are so many different models that can classify data. Why use this one? This is probably the best classifier if you know the data points are easily separable. Also, it can be extended by using kernel

tricks, so try using the different kernels like Radial Basis Function (RBF).

11.9 Code Example

Check out our code, linear_svm.py to learn how to implement a linear SVM using Python's Scikit-learn library. More information about Scikit-Learn can be found here.

linear_svm.py, Classifies a set of data on breast cancer, loaded from Scikit-Learn's dataset library. The program will take the data and plot them on a graph, then use the SVM to create a hyperplane to separate the data. It also circles the support vectors that determine the hyperplane. The output should look like this:



The green points are classified as benign. The red points are classified as malignant.

This loads the data from the Scikit-Learn's dataset library. You can change the data to whatever you would like. Just make sure you have, data points and an array of targets to classify those data points.

```
dataCancer = load_breast_cancer()
data = dataCancer.data[:, :2]
target = dataCancer.target
```

You can also change the kernel to 'rbf' or 'polynomial'. This will create a different hyperplane to classify the data. You can change it here in the code:

```
model = svm.SVC(kernel = 'linear', C = 10000)
model.fit(data, target)
```

11.10 References

1. https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code/
2. https://stackabuse.com/implementing-svm-and-kernel-svm-with-pythons-scikit-learn/

3. https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47
4. https://towardsdatascience.com/https-medium-com-pupalerushikesh-svm-f4b42800e989
5. https://towardsdatascience.com/support-vector-machines-svm-c9ef22815589

CHAPTER 12

Clustering

12.1 Overview

In previous modules, we talked about supervised learning topics. We are now ready to move on to **unsupervised learning** where our goals will be much different. In supervised learning, we tried to match inputs to some existing patterns. For unsupervised learning, we will try to discover patterns in raw, unlabeled data sets. We saw classification problems come up often in supervised learning and we will now examine a similar problem in unsupervised learning: **clustering**.

12.2 Clustering

Clustering is the process of grouping similar data and isolating dissimilar data. We want the data points in clusters we come up with to share some common properties that separate them from data points in other clusters. Ultimately, we'll end up with a number of groups that meet these requirements. This probably sounds familiar because on the surface it sounds a lot like classification. But be aware that clustering and classification solve two very different problems.

Clustering is used to identify potential groups in a data set while classification is used to match an input to an existing group.

12.3 Motivation

Clustering is a very useful technique for solving problems that commonly arise in unsupervised learning. With clustering, we can find underlying patterns in a data set by grouping similar data points. Consider the case of a toy manufacturer. The toy manufacturer makes a lot of products and happens to have a diverse consumer base. It could be useful for the manufacturer to identify groups that buy particular products so it can personalize advertisements. Targeted advertising is a common desire in marketing and clustering helps identify demographics. When we want to identify potential group structures in a raw data set, clustering is a good tool to use.

12.4 Methods

Because clustering just provides an interpretation of a data set, there are many ways to go about implementing it. We might consider the distance between data points when deciding on clusters. We could also consider how dense data points are in a region to determine clusters. For this module, we will analyze two of the more common and popular methods: **K-Means** and **Hierarchical**. In both cases, we will use the data set in *Figure 1* for analysis.

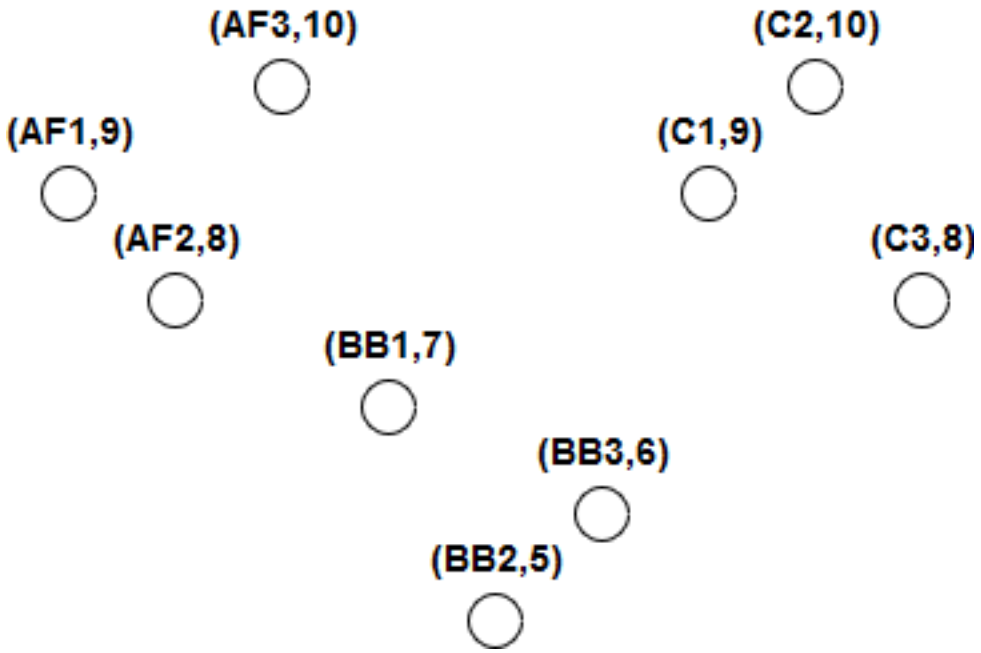


Fig. 1: **Figure 1. A data set to use for clustering**

This data set represents product data for a toy manufacturer. The manufacturer sells 3 products with 3 variants each to young children between the ages of 5 and 10. These products are action figures, building blocks, and cars. The manufacturer has also noted which age group buys the most of each product. Each point in the data set represents one of the toys and the age group that buys the most of them.

12.4.1 K-Means

K-Means clustering attempts to divide a data set into K clusters using an iterative process. The first step is choosing a center point for each cluster. This center point does not need to correspond to an actual data point. The center points could be chosen at random or we could pick them if we have a good guess of where they should be. In the code below, the center points are chosen using the k-means++ method which is designed to speed up convergence. Analysis of this method is beyond the scope of this module but for additional initial options in sklearn, check here.

The second step is assigning each data point to a cluster. We do this by measuring the distance between a data point and each center point and choosing the cluster whose center point is the closest. This step is illustrated in *Figure 2*.

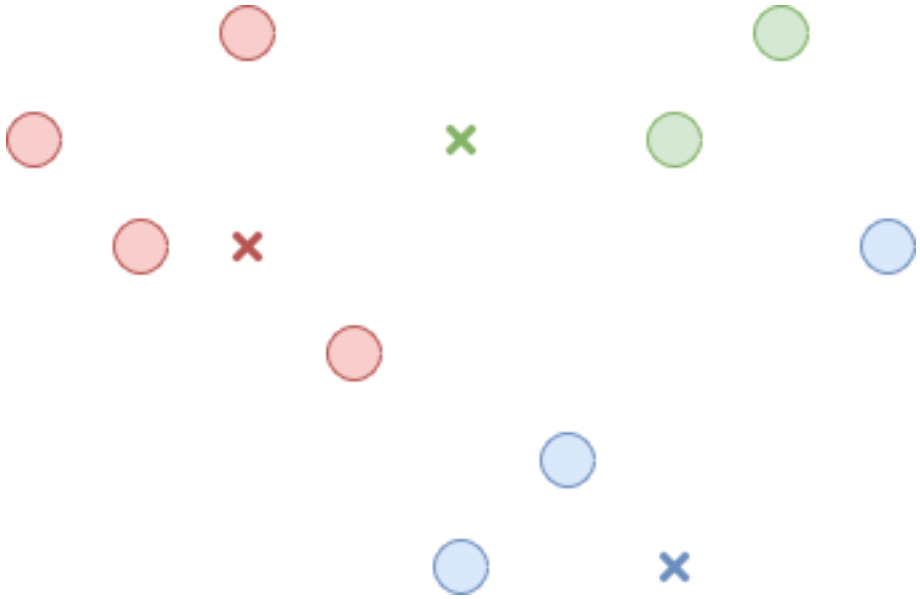


Fig. 2: **Figure 2. Associate each point with a cluster**

Now that all the data points belong to a cluster, the third step is recomputing the center point of each cluster. This is just the average of all the data points belonging to the cluster. This step is illustrated in *Figure 3*.

Now we just repeat the second and third step until the centers stop changing or only change slightly between iterations. The result is K clusters where data points are closer to their cluster's center than any other cluster's center. This is illustrated in *Figure 4*.

K-Means clustering requires us to input the number of expected clusters which isn't always easy to determine. It can also be inconsistent depending on where we choose the starting center points in the first step. Over the course of the process, we may end up with clusters that appear to be optimized but may not be the best overall solution. In *Figure 4*, we end with a red data point that is equally far from the red center and the blue center. This stemmed from our initial center choices. In contrast, *Figure 5* shows another result we may have reached given different starting centers and looks a little better.

On the other hand, K-Means is very powerful because it considers the entire data set at each step. It is also fast because we're only ever computing distances. So if we want a fast technique that considers the whole data set and we have some knowledge of what the underlying groups might look like, K-Means is a good choice.

The relevant code is available in the clustering_kmeans.py file.

In the code, we create the simple data set to use for analysis. Setting up the clustering is very simple and requires one line of code:

```
kmeans = KMeans(n_clusters=3, random_state=0).fit(x)
```

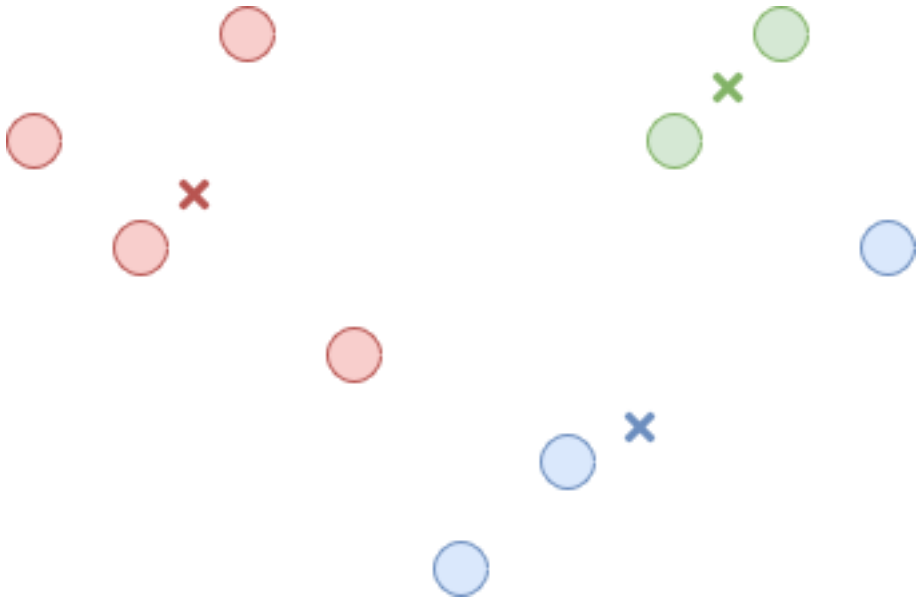


Fig. 3: **Figure 3. Find the new center for each cluster**

Fig. 4: **Figure 4. The final clusters**

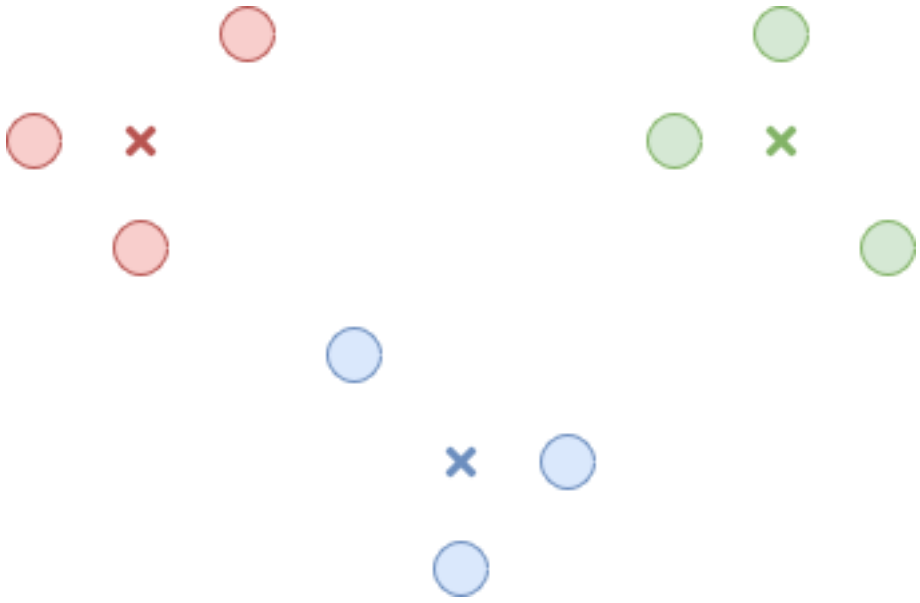


Fig. 5: **Figure 5. An alternative set of clusters**

The `n_clusters` parameter was chosen to be 3 because there appears to be 3 clusters in out data set. The `random_state` parameter is just there to give a consistent result each time you run the code. The rest of the code is to display the final plot shown in *Figure 6*.

The clusters are color coded, the 'x's represent cluster centers, and the dotted lines represent cluster boundaries.

12.4.2 Hierarchical

Hierarchical clustering imagines the data set as a hierarchy of clusters. We could start by making one giant cluster out of all the data points. This is illustrated in *Figure 7*.

Inside of this cluster, we find the two least similar sub-clusters and split them. This can be done by using an algorithm to maximize the inter-cluster distance. This is just the smallest distance between a node from one cluster and a node from the other cluster. This is illustrated in *Figure 8*.

We continue to split the sub-clusters until every data point belongs to its own cluster or until we decide to stop. If we start from one giant cluster and break it down into successively smaller clusters, it is called **top-down** or **divisive** clustering. Alternatively, we could start by considering a cluster for every data point. The next step would be to combine the two closest clusters into a larger cluster. This can be done by finding the distance between every cluster and choosing the pair with the least distance between them. We would continue this process until we had a single cluster. This method of combining clusters is called **bottom-up** or **agglomerative** clustering. At any point in these two methods, we can stop when the clusters look appropriate.

Unlike K-Means, Hierarchical clustering is relatively slow so it doesn't scale as well to large data sets. On the bright side, Hierarchical clustering is more consistent when you run it multiple times and doesn't require you to know the number of expected clusters.

The relevant code is available in the clustering_hierarchical.py file.

In the code, we create the simple data set to use for analysis. Setting up the clustering is very simple and requires one line of code:

```
hierarchical = AgglomerativeClustering(n_clusters=3).fit(x)
```

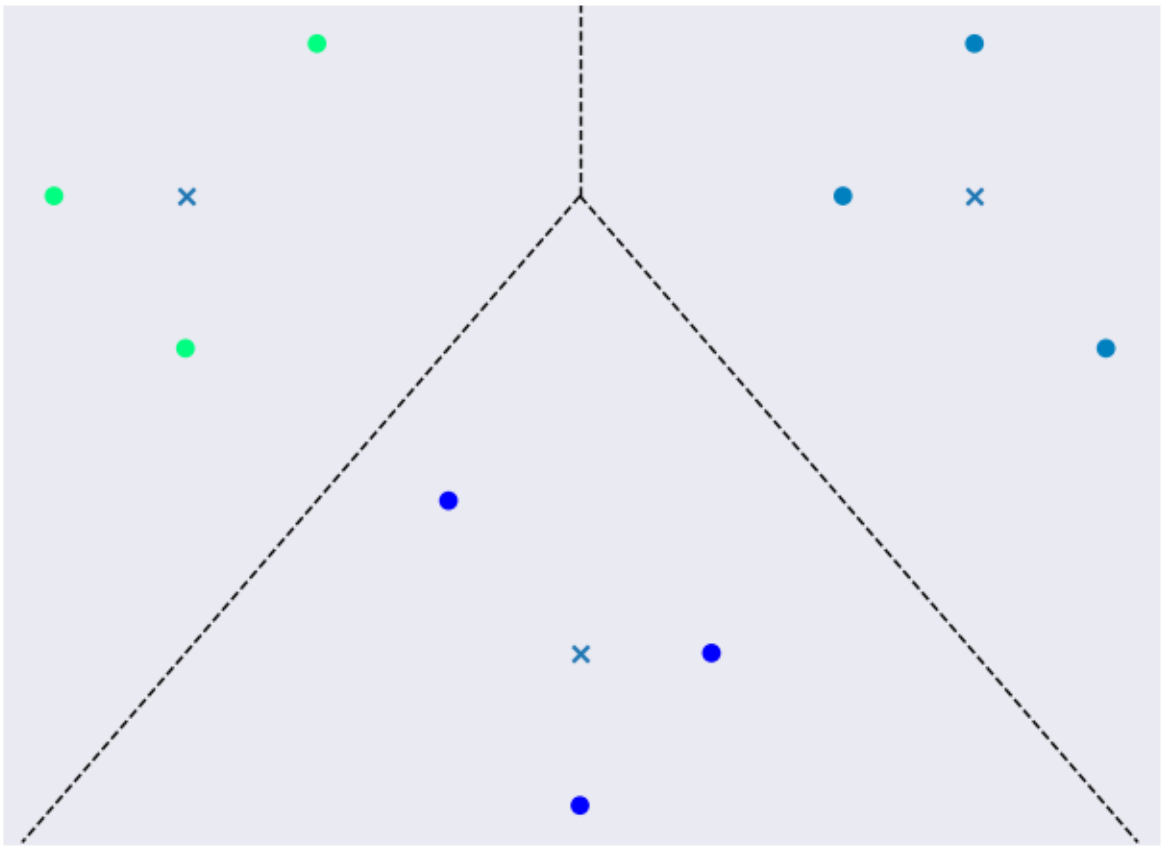


Fig. 6: **Figure 6. A final clustered data set**

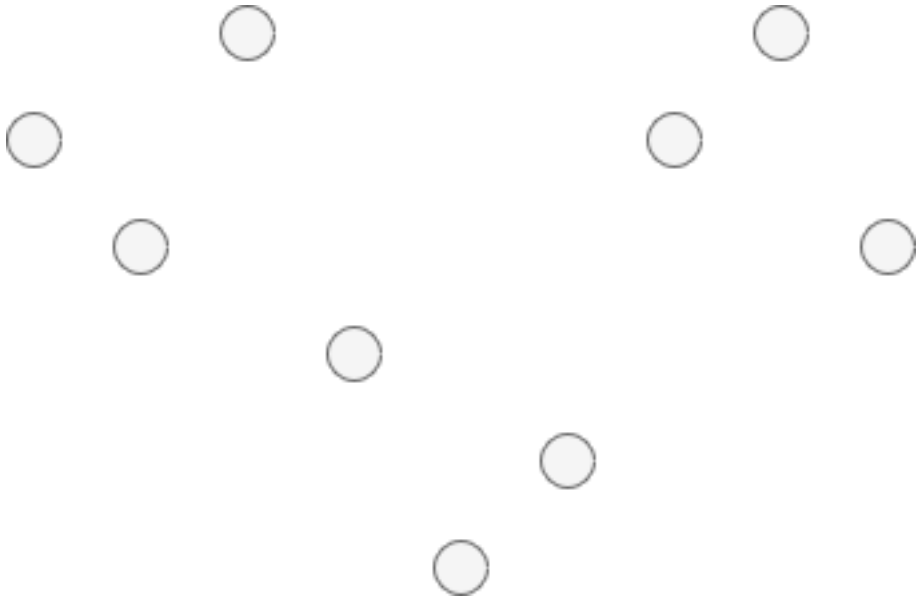


Fig. 7: **Figure 7. One giant cluster in the data set***

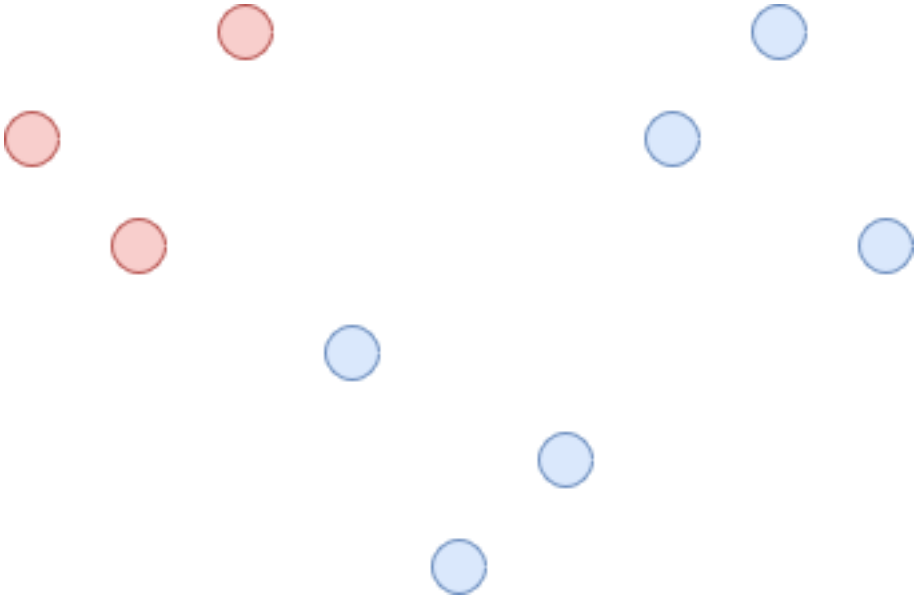


Fig. 8: **Figure 8. The giant cluster is split into 2 clusters**

The `n_clusters` parameter was chosen to be 3 because there appears to be 3 clusters in out data set. If we didn't already know this, we could try out different values and see which one worked the best. The rest of the code is to display the final plot shown in *Figure 9*.

The clusters are color coded and large clusters are surrounded with a border to show which data points belong to them.

12.5 Summary

In this module, we learned about clustering. Clustering allows us to discover patterns in a raw data set by grouping similar data points. This is a common desire in unsupervised learning and clustering is a popular technique. You may have noticed that the methods discussed above were relatively simple compared to some of the more math-heavy descriptions in previous modules. These methods are simple but powerful. For example, we were able to determine clusters in the toy manufacturer example that could be used for targeted advertising. This is a very useful result for businesses and it only took us a few lines of code. By developing a good understanding of clustering, you are setting yourself up for success in the machine learning world.

12.6 References

1. https://www.analyticsvidhya.com/blog/2016/11/an-introduction-to-clustering-and-different-methods-of-clustering/
2. https://medium.com/datadriveninvestor/an-introduction-to-clustering-61f6930e3e0b
3. https://medium.com/predict/three-popular-clustering-methods-and-when-to-use-each-4227c80ba2b6
4. https://towardsdatascience.com/the-5-clustering-algorithms-data-scientists-need-to-know-a36d136ef68
5. https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html

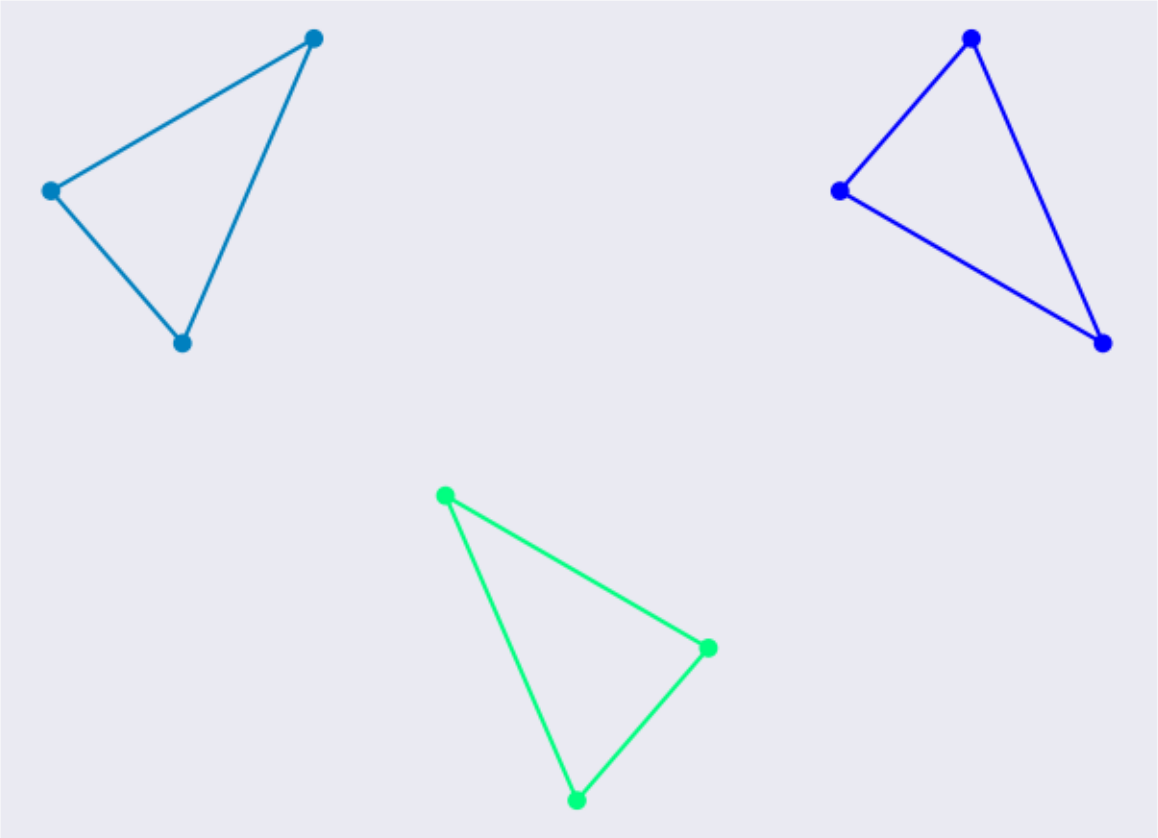


Fig. 9: **Figure 9. A final clustered data set**

CHAPTER 13

Principal Component Analysis

13.1 Introduction

Principal component analysis is one technique used to take a large list of interconnected variables and choose the ones that best suit a model. This process of focusing in on only a few variables is called **dimensionality reduction**, and helps reduce complexity of our dataset. At its root, principal component analysis *summarizes* data.

13.2 Motivation

Principal component analysis is extremely useful for deriving an overall, linearly independent, trend for a given dataset with many variables. It allows you to extract important relationships out of variables that may or may not be related. Another application of principal component analysis is for display - instead of representing a number of different variables, you can create principal components for just a few and plot them.

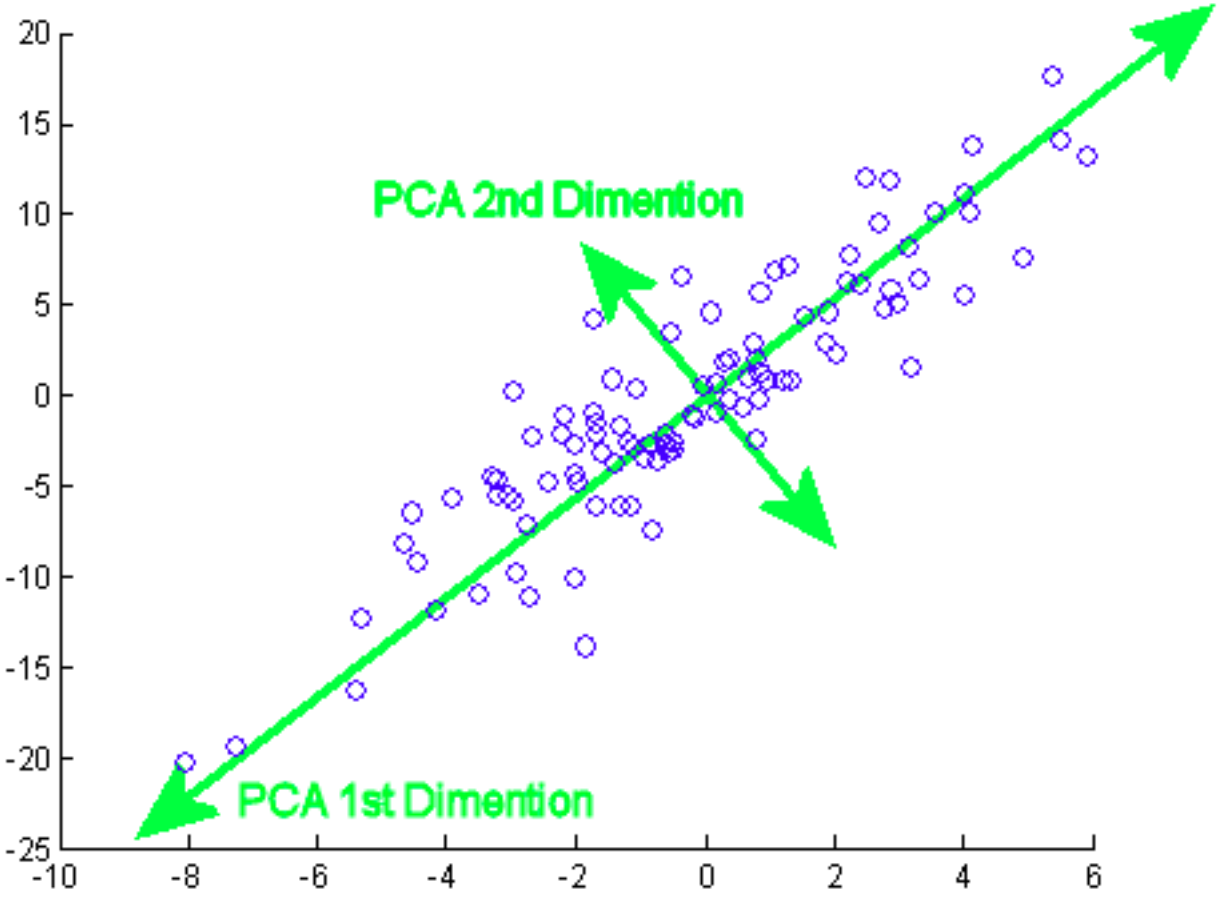


Fig. 1: Ref: https://stats.stackexchange.com/questions/2691/making-sense-of-principal-component-analysis-eigenvectors-eigenvalues

13.3 Dimensionality Reduction

There are two types of dimensionality reduction: feature elimination and feature extraction.

Feature elimination simply involves pruning features from a dataset we deem unnecessary. A downside of feature elimination is that we lose any potential information gained from the dropped features.

Feature extraction, however, creates new variables by combining existing features. At the cost of some simplicity or interpretability, feature extraction allows you to maintain all important information held within features.

Principal component analysis deals with feature extraction (rather than elimination) by creating a set of independent variables called principal components.

13.4 PCA Example

Principal component analysis is performed by considering all of our variables and calculating a set of direction and magnitude pairs (vectors) to represent them. For example, let's consider a small example dataset plotted below:

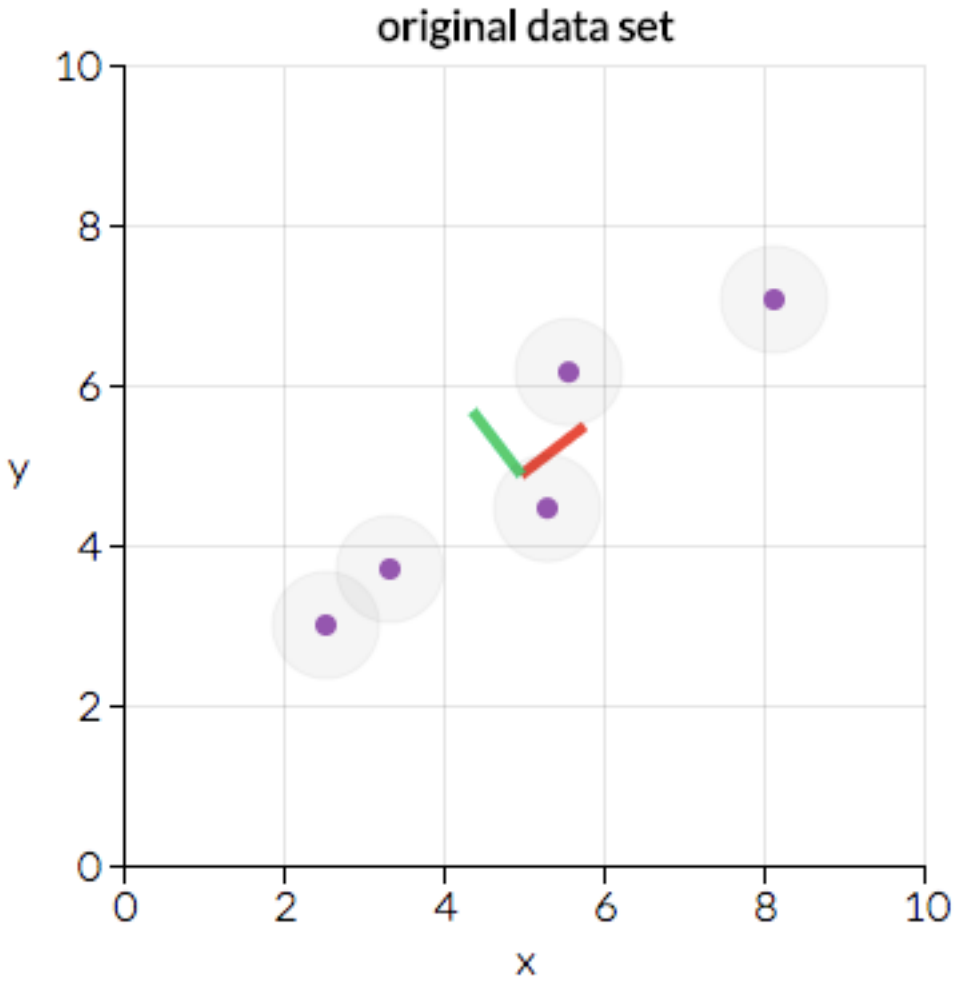


Fig. 2: Ref: https://towardsdatascience.com/a-one-stop-shop-for-principal-component-analysis-5582fb7e0a9c

Here we can see two direction pairs, represented by the red and green lines. In this scenario, the red line has a

greater magnitude as the points are more clustered across a greater distance than with the green direction. Principal component analysis will use the vector with the greater magnitude to transform the data into a smaller feature space, reducing dimensionality. For example, the above graph would be transformed into the following:

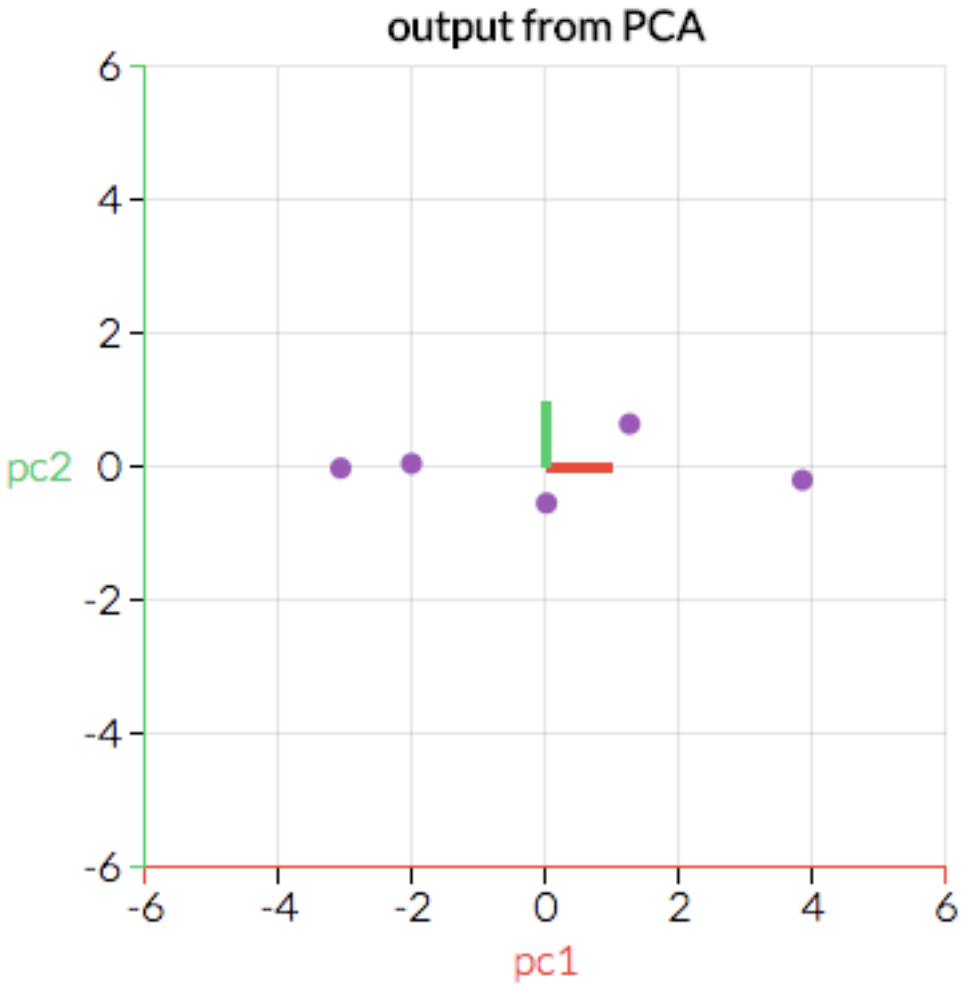


Fig. 3: Ref: https://towardsdatascience.com/a-one-stop-shop-for-principal-component-analysis-5582fb7e0a9c

By transforming our data in this way, we've ignored a feature that is less important to our model - that is, higher variation along the green dimension will have a greater impact on our results than variation along the red.

The mathematics behind principal component analysis are left out of this discussion for brevity, but if you're interested in learning about them we highly recommend visiting the references listed at the bottom of this page.

13.5 Number of Components

In the example above, we took a two-dimensional feature space and reduced it to a single dimension. In most scenarios though, you will be working with far more than two variables. Principal component analysis can be used to just remove a single feature, but it is often useful to reduce several. There are several strategies you can employ to decide how many feature reductions to perform:

1. **Arbitrarily**

This simply involves picking a number of features to keep for your given model. This method is highly dependent on your dataset and what you want to convey. For instance, it may be beneficial to represent your higher-order data on a 2D space for visualization. In this case, you would perform feature reduction until you have two features.

2. **Percent of cumulative variability**

   Part of the principal component analysis calculation involves finding a proportion of variance which approaches 1 through each round of PCA performed. This method of choosing the number of feature reduction steps involves selecting a target variance percentage. For instance, let's look at a graph of cumulative variance at each level of PCA for a theoretical dataset:

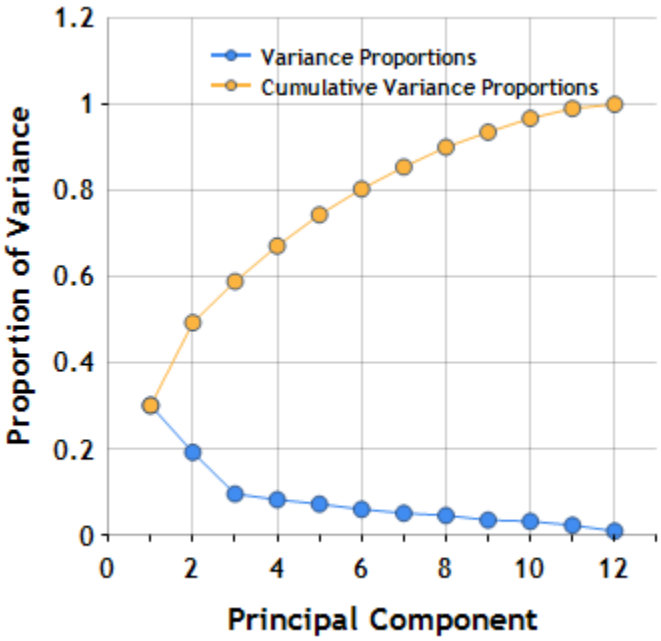


Fig. 4: Ref: https://www.centerspace.net/clustering-analysis-part-i-principal-component-analysis-pca

   The above image is called a scree plot, and is a representation of the cumulative and current proportion of variance for each principal component. If we wanted at least 80% cumulative variance, we would use at least 6 principal components based on this scree plot. Aiming for 100% variance is not generally recommended, as reaching this means your dataset has redundant data.

3. **Percent of individual variability**

   Instead of using principal components until we reach a cumulative percent of variability, we can instead use principal components until a new component wouldn't add much variability. In the plot above, we might choose to use 3 principal components since the next components don't have as strong a drop in variability.

13.6 Conclusion

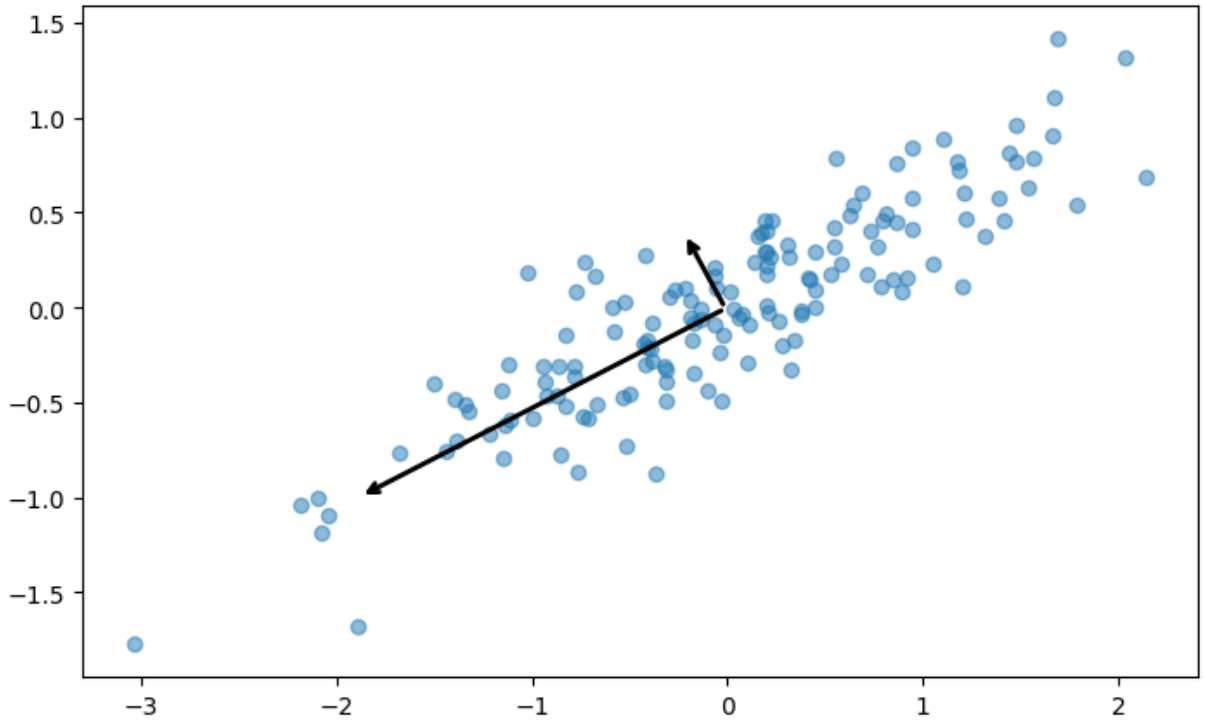
Principal component analysis is a technique to summarize data, and is highly flexible depending on your use case. It can be valuable in both displaying and analyzing a large number of possibly dependent variables. Techniques of performing principal component analysis range from arbitrarily selecting principal components, to automatically finding them until a variance is reached.

13.7 Code Example

Our example code, pca.py, shows you how to perform principal component analysis on a dataset of random x, y pairs. The script goes through a short process of generating this data, then calls sklearn's PCA module:

```
# Find two principal components from our given dataset
pca = PCA(n_components = 2)
pca.fit(points)
```

Each step in the process includes helpful visualizations using matplotlib. For instance, the principal components fitted above are plotted as two vectors on the dataset:



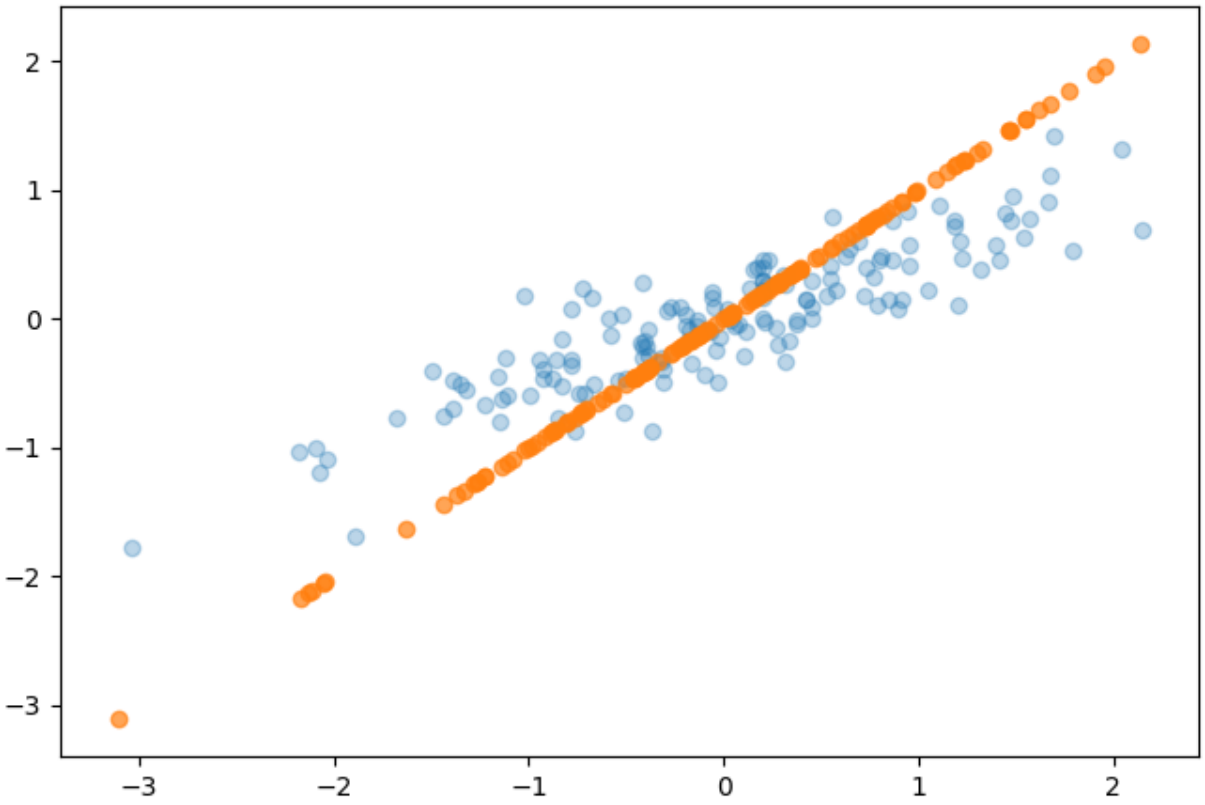
The script also shows how to perform dimensionality reduction, discussed above. In sklearn, this is done by simply calling the transform method once a PCA is fitted, or doing both steps at the same time with fit_transform:

```
# Reduce the dimensionality of our data using a PCA transformation
pca = PCA(n_components = 1)
transformed_points = pca.fit_transform(points)
```

The end result of our transformation is just a series of X values, though the code example performs an inverse transformation for plotting the result in the following graph:

13.8 References

1. http://www.cs.otago.ac.nz/cosc453/student_tutorials/principal_components.pdf
2. https://towardsdatascience.com/a-one-stop-shop-for-principal-component-analysis-5582fb7e0a9c



3. https://towardsdatascience.com/pca-using-python-scikit-learn-e653f8989e60
4. https://en.wikipedia.org/wiki/Principal_component_analysis
5. https://stats.stackexchange.com/questions/2691/making-sense-of-principal-component-analysis-eigenvectors-eigenvalues
6. https://www.centerspace.net/clustering-analysis-part-i-principal-component-analysis-pca

CHAPTER 14

Multi-layer Perceptron

14.1 Overview

A multilayer perceptron (MLP) is a deep, artificial **neural network**. A *neural network* is comprised of layers of nodes which activate at various levels depending on the previous layer's nodes. When thinking about neural networks, it may be helpful to isolate your thinking to a single node in the network.

Multilayer perceptron refers to a neural network with at least three layers of nodes, an input layer, some number of intermediate layers, and an output layer. Each node in a given layer is connected to every node in the adjacent layers. The input layer is just that, it is the way the network takes in data. The intermediate layer(s) are the computational machine of the network, they actually transform the input to the output. The output layer is the way that results are obtained from the neural network. In a simple network where the responses are binary, there would likely be only one node in the output layer, which outputs a probability like in logistic regression.

For a visual look at a neural network in action, play with this website where you can see a number recognition neural network. In this section, the ideas are focused on the "fully connected" layers, so try to think in terms of those.

They require labeled sample data, so they carry out **supervised learning**. For each training sample, nodes activate according to stored weights of the previous layer. During training (and beyond), the weights will not be perfectly accurate, so they will need to change a little bit to meet the desired results. MLPs use a method called *backpropagation* to learn from training data, which we will explore briefly here.

14.2 Motivation

Multilayer perceptron is the basic type of neural network, and should be well understood before moving on to more advanced models. By examining MLPs, we should be able to avoid some of the complications that come up in more advanced topics in deep learning, and establish a baseline of knowledge.

This is not to undervalue the topic, as even simple networks can achieve great results. It was proven that a network with a single hidden layer could approximate any continuous function.

14.3 What is a node?

A node is a single unit in a neural network. Nodes **activate** at different levels depending on a weighted sum of the previous layer's nodes. In practice, the actual activation is the result of a **sigmoid function** applied to this result, but we will skip over that detail here for simplicity. Thinking of the output in this way won't lose any of the magic of neural networks, while avoiding some painful details. In MLPs, nodes activate based on **all** of the nodes in the previous layer.

In this example, let's focus on the single-node layer, which is that way for example purposes. Each line represents the weights of the nodes in the previous layer. The sum of the weights of each connection multiplied by the activation of the connected node results in the activation of our node. The key here is the weights, since they determine the output of the node. Remember that nodes only take input from the previous layer, so the weights are the only differentiator of nodes in the same layer.

14.4 What defines a multilayer perceptron?

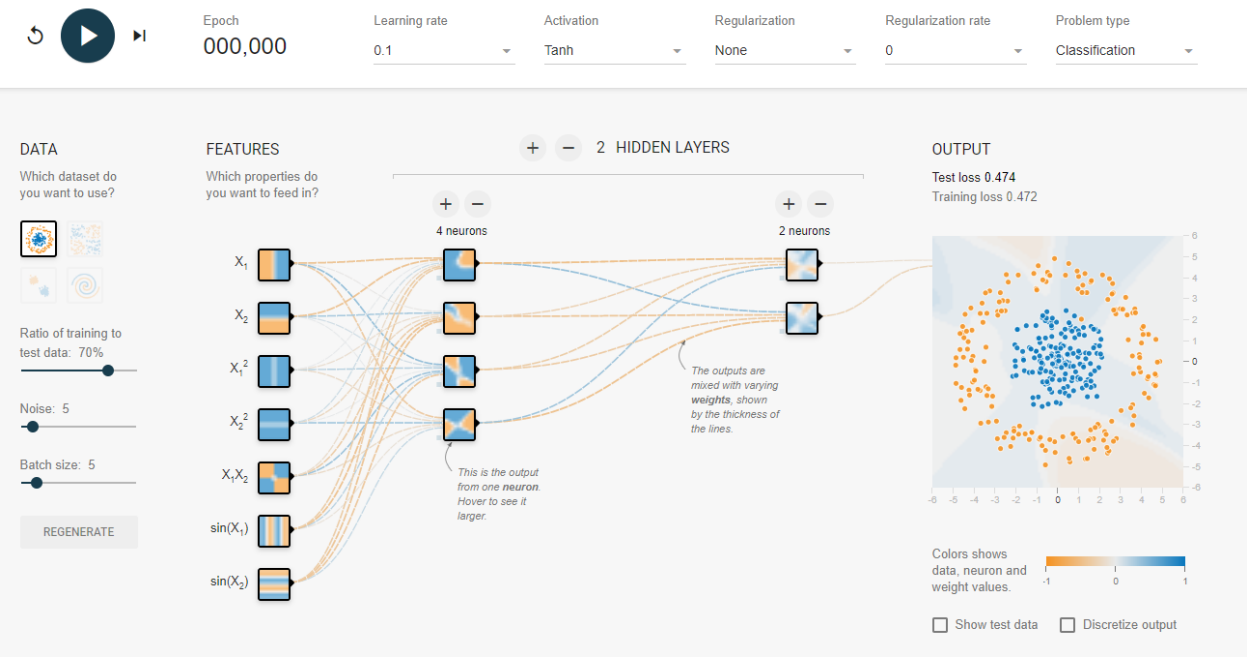
A Multilayer Perceptron (MLP) is a type of **feedforward** neural network which is characterized by an input layer, some number of intermediate layers, and an output layer, which are **fully connected**. An MLP uses **backpropagation** for training. The term **feedforward** refers to the layered architecture in the network, specifically that there are no cycles in the network. The layer structure ensures no cycles exists, as layers are only allowed to have weights from the directly previous layer. The term **fully connected** refers to the fact that in MLP, all nodes in a given layer have a weight to all of the nodes in the previous layer.

14.5 What is backpropagation?

When training a neural network, the expected output is a level of activation for each node in the output layer. From that information and the actual activation, we can find the *cost* at each node, and adjust the weights accordingly. The idea of backpropagation is to adjust the weights that determine each node's activation based on the cost.

To keep the idea here high-level, we will avoid the details on how the exact math works, and focus on the big picture. If you would like to see the math, check out this article.

Take a look at this screenshot taken from the tensorflow test site linked earlier. Here, we are training a neural network to classify the blue dots and the orange dots. The choices made for the nodes here are arbitrary, and we encourage you to mess around with them.



To talk about backpropagation, let's consider what this network will do the first time step. The network will test some training data in the network, expecting to see full activation on the (hidden) correct output node and no activation on the wrong one. When the model is not right, it will look from the output backwards to find out how wrong it was. Then, it will change the weights accordingly, so weights that were way off will change more than those that were close. In these early steps, it will have a high **learning rate**, making the weights more volatile. After a few iterations, it will be much more stable as it should need smaller adjustments. With that in mind, let's move forward one time step.

Now, the network has a vague idea of how to classify the data. It has a loose circle which will become more clear as we go on. Let's jump forward a few more steps.

As you can see, the model has developed much better performance, classifying most of the points accurately. At this point, the network slows the **learning rate**, since it has gone through enough iterations to be somewhat successful.

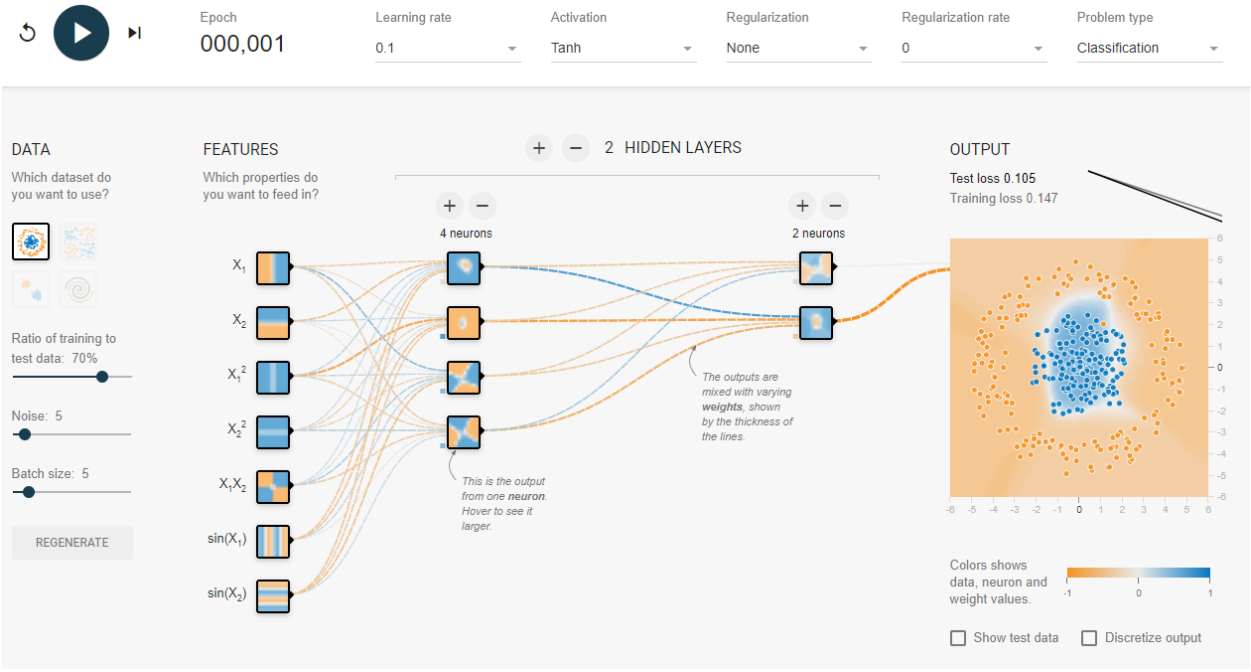
14.6 Summary

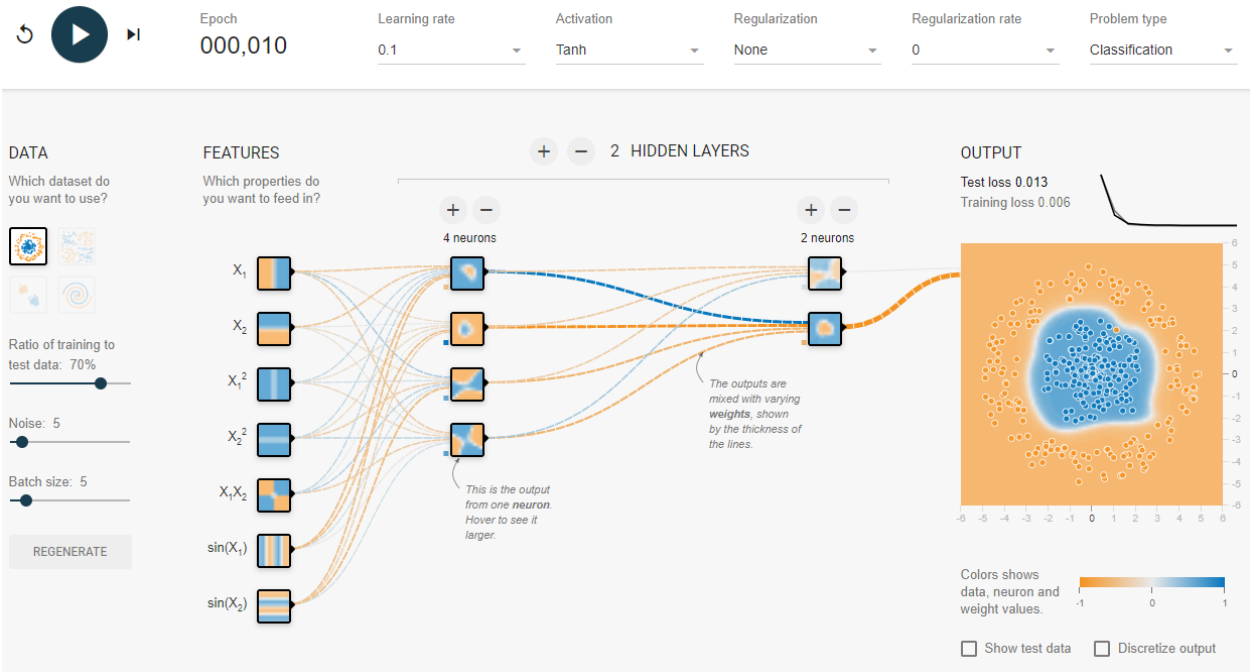
In this section, we learned about the multilayer perceptron (MLP) class of neural networks, and a little bit about neural networks as a whole. We touched on what a *node* is, and what it knows about the things going on around it. We discussed how a network learns from training data, specifically using *backpropagation*. We also looked into what defines an MLP network, and how they differ from other neural networks.

14.7 Further Resources

If you wish to learn more about the topic of neural networks, we recommend this playlist by 3Blue1Brown on YouTube.

The playlist covers a more visual approach to neural networks, and can help you fill in some of the details on neural networks.


Epoch
000,001
Learning rate
0.1
Activation
Tanh
Regularization
None
Regularization rate
0
Problem type
Classification
DATA
Which dataset do you want to use?
Ratio of training to test data: 70%
Noise: 5
Batch size: 5
REGENERATE
FEATURES
Which properties do you want to feed in?
2 HIDDEN LAYERS
4 neurons
2 neurons
X1
X2
X1²
X2²
X1X2
sin(X1)
sin(X2)
This is the output from one neuron. Hover to see it larger.
The outputs are mixed with varying weights, shown by the thickness of the lines.
OUTPUT
Test loss 0.105
Training loss 0.147
Colors shows data, neuron and weight values.
Show test data
Discretize output


Epoch
000,010
Learning rate
0.1
Activation
Tanh
Regularization
None
Regularization rate
0
Problem type
Classification
DATA
Which dataset do you want to use?
Ratio of training to test data: 70%
Noise: 5
Batch size: 5
REGENERATE
FEATURES
Which properties do you want to feed in?
2 HIDDEN LAYERS
4 neurons
2 neurons
X1
X2
X1²
X2²
X1X2
sin(X1)
sin(X2)
This is the output from one neuron. Hover to see it larger.
The outputs are mixed with varying weights, shown by the thickness of the lines.
OUTPUT
Test loss 0.013
Training loss 0.006
Colors shows data, neuron and weight values.
Show test data
Discretize output

14.8 References

1. https://playground.tensorflow.org/
2. https://en.wikipedia.org/wiki/Universal_approximation_theorem
3. https://www.techopedia.com/definition/20879/multilayer-perceptron-mlp
4. http://neuralnetworksanddeeplearning.com/chap2.html
5. https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi
6. http://cs231n.stanford.edu/

CHAPTER 15

Convolutional Neural Networks

15.1 Overview

In the last module, we started our dive into deep learning by talking about multi-layer perceptrons. In this module, we will learn about **convolutional neural networks** also called **CNNs** or **ConvNets**. CNNs differ from other neural networks in that sequential layers are not necessarily fully connected. This means that a subset of the input neurons may only feed into a single neuron in the next layer. Another interesting feature of CNNs is their inputs. With other neural networks we might use vectors as inputs, but with CNNs we are typically working with images and other objects with many dimensions. *Figure 1* shows some sample images that are each 6 pixels by 6 pixels. The first image is colored and has three channels for red, green, and blue values. The second image is black-and-white and only has one channel for gray values

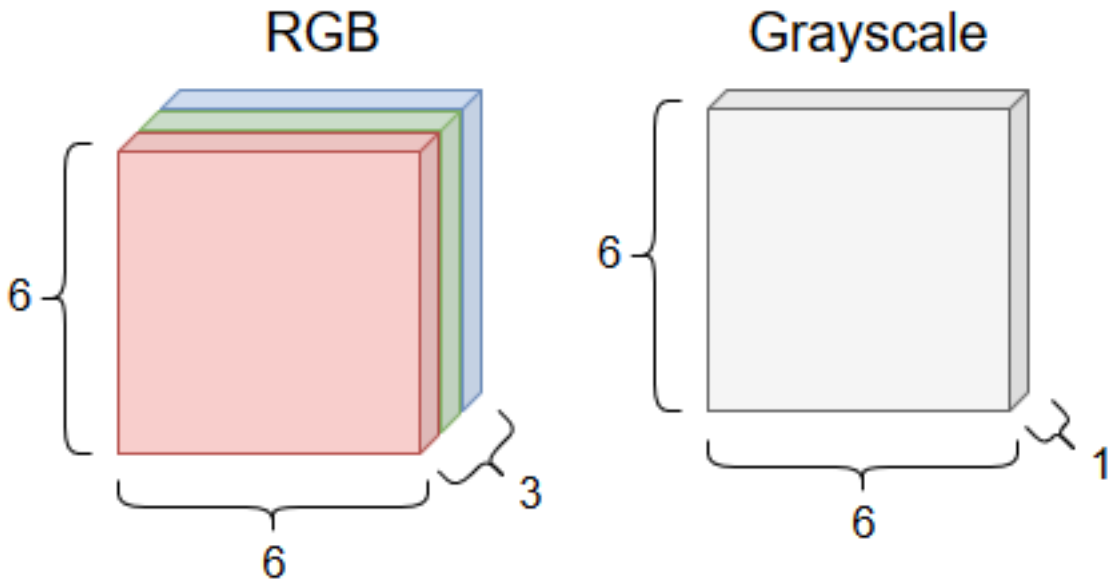


Fig. 1: **Figure 1. Two sample images and their color channels**

15.2 Motivation

CNNs are widely used in computer vision where we are trying to analyze visual imagery. CNNs can also be used for other applications such as natural language processing. We will be focusing on the former case here because it is one of the most common applications of CNNs.

Because we assume that we're working with images, we can design our architecture so that it specifically does a good job at analyzing images. Images have heights, depths, and one or more channels for color. In an image, there might be lines and edges that make up shapes as well as more complex structures such as cars and faces. We will potentially need to identify a large set of relevant features in order to properly classify an image. But just identifying individual features in an image usually isn't enough. Say we have an image that may or may not be a face. If we saw three noses, an eye, and an ear, we probably wouldn't call it a face even though those are common features of a face. So then we must also care about where features are located in the image and their proximity to other features. This is a lot of information to keep track of! Fortunately, the architecture of CNNs will cover a lot of these requirements.

15.3 Architecture

The architecture of a CNN can be broken down into an input layer, a set of hidden layers, and an output layer. These are shown in *Figure 2*.

The hidden layers are where the magic happens. The hidden layers will break down our input image in order to identify features present in the image. The initial layers focus on low-level features such as edges while the later layers progressively get more abstract. At the end of all the layers, we have a fully connected layer with neurons for each of our classification values. What we end up with is a probability for each of the classification values. We choose the classification with the highest probability as our guess for what the image show.

Below, we will talk about some types of layers we might use in our hidden layers. Remember that sequential layers are not necessarily fully connected with the exception of the final output layer.

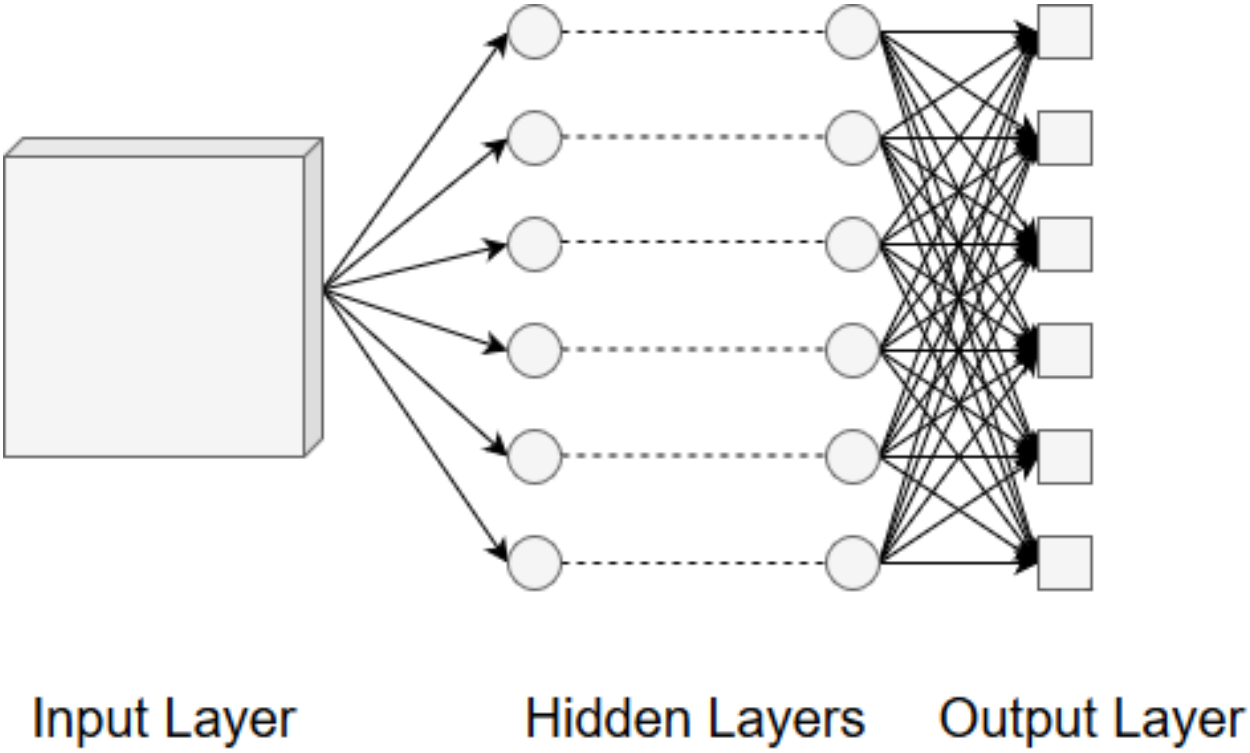


Fig. 2: **Figure 2. The layers of a CNN**

15.3.1 Convolutional Layers

The first type of layer we will discuss is called a **convolutional layer**. The convolutional description comes from the concept of a convolution in mathematics. Roughly, a convolution is some operation that acts on two input functions and produces an output function that combines the information present in the inputs. The first input will be our image and the second input will be some sort of filter such as a blur or sharpen. When we combine our image with the filter, we extract some information about the image. This process is shown in *Figure 3*. This is precisely how the CNN will go about extracting features.

Fig. 3: **Figure 3. An image before and after filtering**

In the human eye, a single neuron is only responsible for a small region of our field of view. It is through many neurons with overlapping regions that we are able to see the world. CNNs are similar. The neurons in a convolutional layer are

only responsible for analyzing a small region of the input image but overlap so that we ultimately analyze the whole image. Let's examine that filter concept we mentioned above.

The **filter** or **kernel** is one of the functions used in the convolution. The filter will likely have a smaller height and width than the input image and can be thought of as a window sliding over the image. *Figure 4* shows a sample filter and the region of the image it will interact with in the first step of convolution.

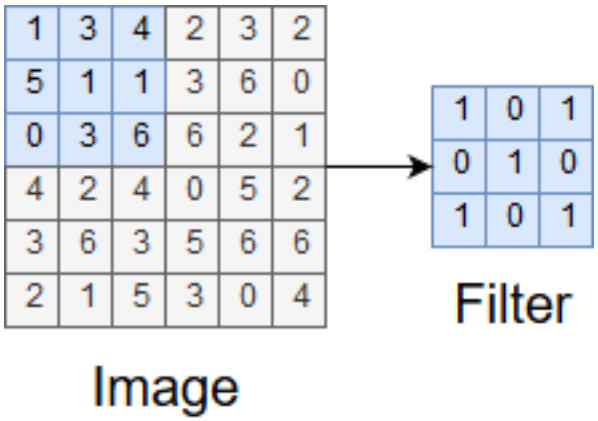


Fig. 4: **Figure 4. A sample filter and sample window of an image**

As the filter moves across the image, we are calculating values for the convolution output called a **feature map**. At each step, we multiply each entry in the image sample and filter elementwise and sum up all the products. This becomes an entry in the feature map. This process is shown in *Figure 5*.

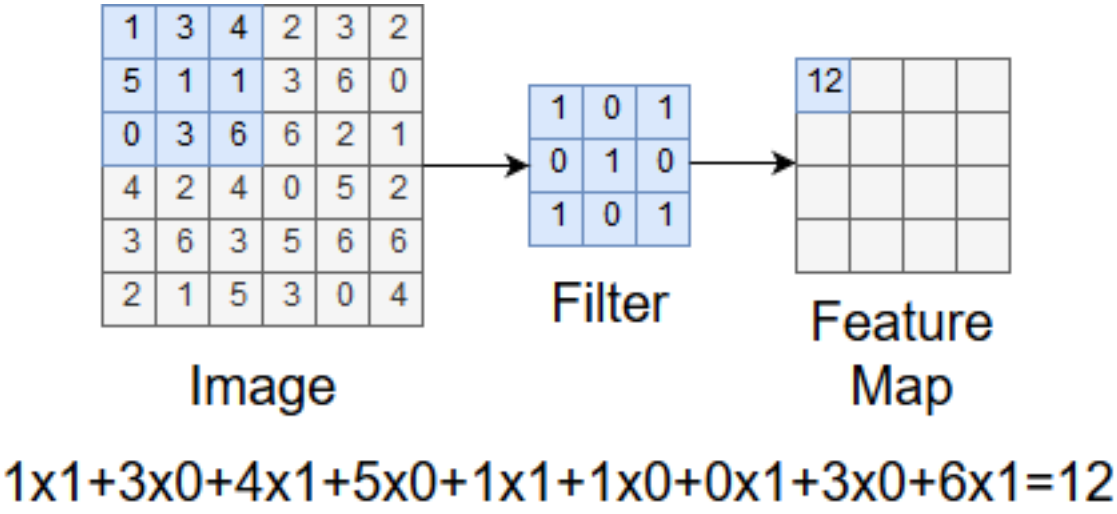


Fig. 5: **Figure 5. Calculating an entry in the feature map**

After the window traverses the entire image, we have the complete feature map. This is shown in *Figure 6*.

In the example above, we moved the filter one unit horizontally or one unit vertically from some previous position. This value is called the **stride**. We could have used other values for the stride but using one everywhere tends to produce the best results.

You may have noticed that the feature map we ended up with had a smaller height and width than the original image sample. This is a result of the way we moved the filter around the sample. If we wanted the feature map to have the same height and width, we could **pad** the sample. This involves adding zero entries around the sample so that moving the filter keeps the dimensions of the original sample in the feature map. *Figure 7* illustrates this process.

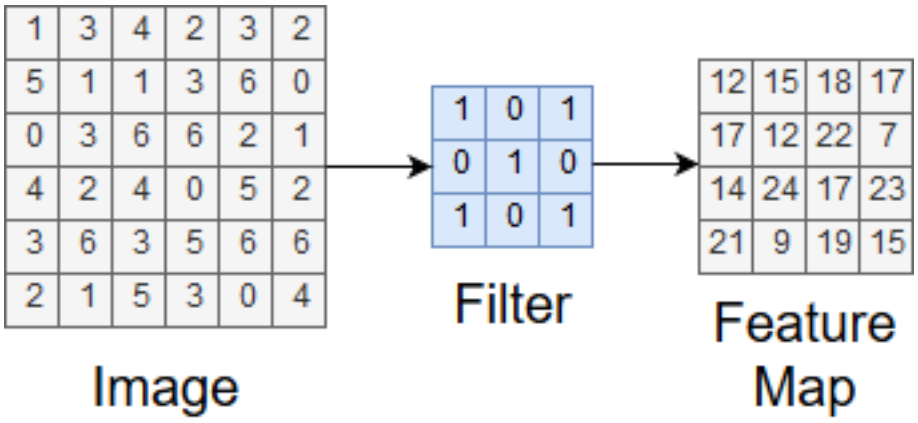


Fig. 6: **Figure 6. The complete feature map**

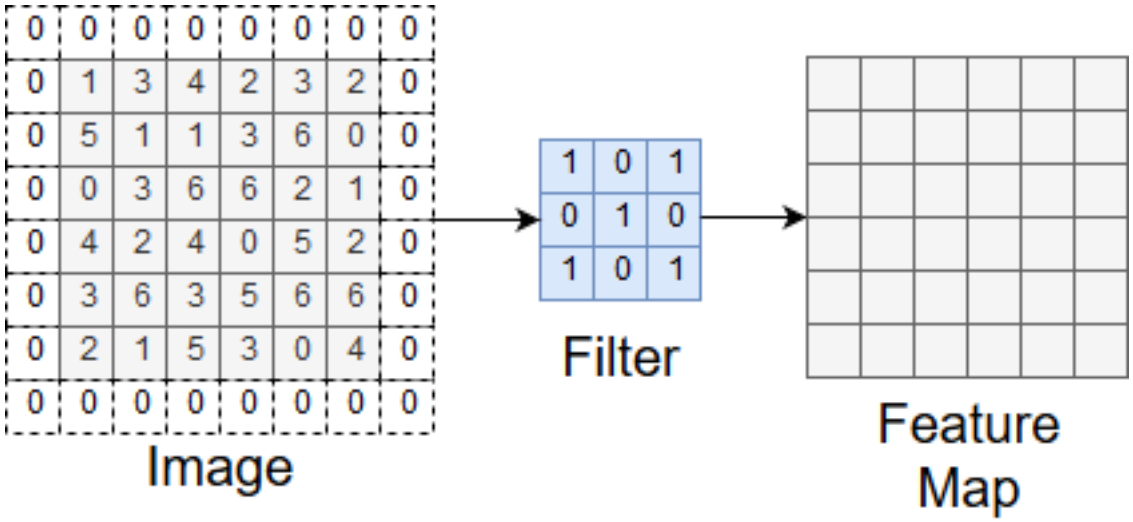


Fig. 7: **Figure 7. Padding before applying a filter**

A feature map represents one type of feature we're analyzing the image for. Often, we want to analyze the image for a bunch of features so we end up with a bunch of feature maps! The output of the convolution layer is a set of feature maps. *Figure 8* shows the process of going from an image to the resulting feature maps.

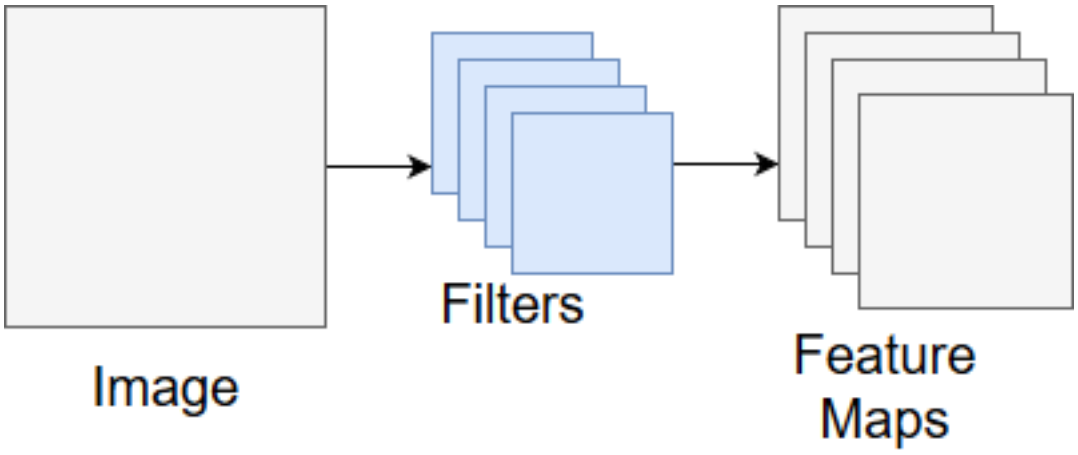


Fig. 8: **Figure 8. The output of a convolutional layer**

After a convolutional layer, it is common to have a **ReLU** (rectified linear unit) layer. The purpose of this layer is to introduce non-linearity into the system. Basically, real-world problems are rarely nice and linear so we want our CNN to account for this when it trains. A good explanation of this layer requires math that we don't expect you to know. If you are curious about the topic, you can find an explanation here.

15.3.2 Pooling Layers

The next type of layer we will cover is called a **pooling layer**. The purpose of pooling layers are to reduce the spatial size of the problem. This in turn reduces the number of parameters needed for processing and the total amount of computation in the CNN. There are several options for pooling but we will cover the most common approach, **max pooling**.

In max pooling, we slide a window over the input and take the max value in the window at each step. This process is shown in *Figure 9*.

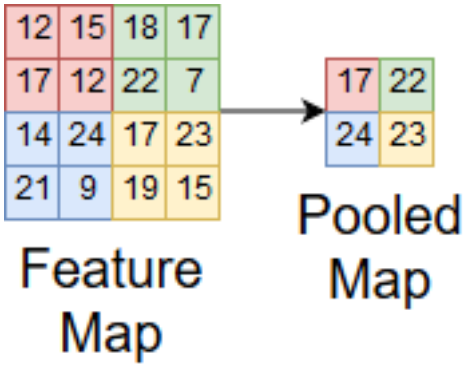


Fig. 9: **Figure 9. Max pooling on a feature map**

Max pooling is good because it maintains important features about the input, reduces noise by ignoring small values, and reduces the spatial size of the problem. We can use these after convolutional layers to keep the computation of problems manageable.

15.3.3 Fully Connected Layers

The last type of layer we will discuss is called a **fully connected layer**. Fully connected layers are used to make the final classification in the CNN. They work exactly like they do in other neural networks. Before moving to the first fully connected layer, we must flatten our input values into a one-dimensional vector that the layer can interpret. *Figure 10* shows a simple example of converting a multi-dimensional input into a one-dimensional vector.

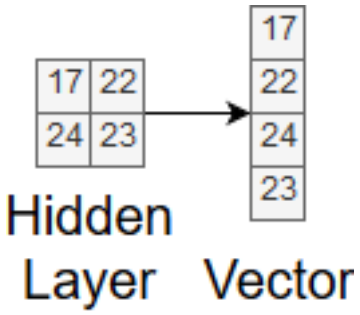


Fig. 10: **Figure 10. Flattening input values**

After doing this, we may have several fully connected layers before the final output layer. The output layer uses some function, such as softmax, to convert the neuron values into a probability distribution over our classes. This means that the image has a certain probability for being classified as one of our classes and the sum of all those probabilities equals one. This is clearly visible in *Figure 11*.

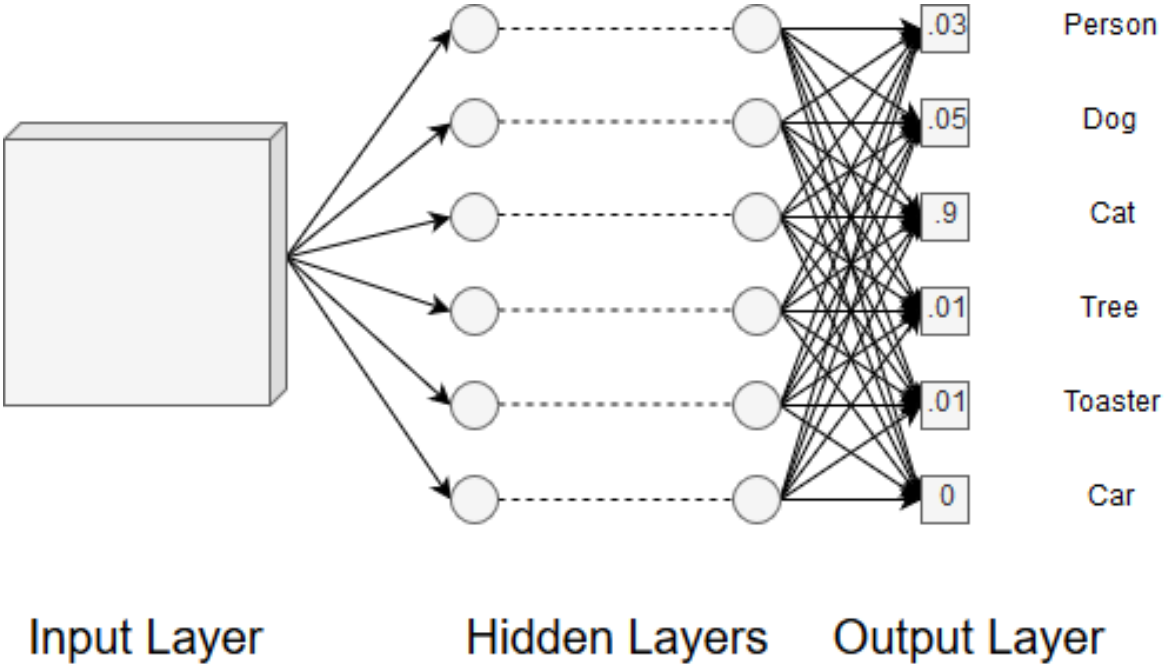


Fig. 11: **Figure 11. The final probabilistic outputs**

15.4 Training

Now that we have the architecture in place for CNNs we can move on to training. Training a CNN is pretty much exactly the same as training a normal neural network. There is some added complexity due to the convolutional layers

but the strategies for training remain the same. Techniques, such as gradient descent or backpropagation, can be used to train filter values and other parameters in the network. As with all the other training we have covered, having a large training set will improve the performance of the CNN. The problem with training CNNs and other deep learning models is that they are much more complex than the models we covered in earlier modules. This results in training being much more computationally expensive to the point where we would need specialized hardware like GPUs to run our code. However, we get what we pay for because deep learning models are much more powerful than the models covered in earlier modules.

15.5 Summary

In this module, we learned about convolutional neural networks. CNNs differ from other neural networks because they usually take images as input and can have hidden layers that are not fully connected. CNNs are powerful tools widely used in image classification applications. By using a variety of hidden layers, we can extract features from an image and use them to probabilistically guess a classification. CNNs are also complex models and understanding how they work can be an intimidating task. We hope that the information presented gives you a better understanding of how CNNs work so that you can continue to learn about them and deep learning.

15.6 References

1. https://towardsdatascience.com/convolutional-neural-networks-for-beginners-practical-guide-with-python-and-keras-dc688ea90
2. https://medium.com/technologymadeeasy/the-best-explanation-of-convolutional-neural-networks-on-the-internet-fbb8b1ad5df8
3. https://medium.freecodecamp.org/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050
4. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53
5. https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/
6. https://www.kaggle.com/dansbecker/rectified-linear-units-relu-in-deep-learning
7. https://en.wikipedia.org/wiki/Convolutional_neural_network#ReLU_layer

CHAPTER 16

Autoencoders

16.1 Autoencoders and their implementations in TensorFlow

In this post, you will learn the notion behind Autoencoders as well as how to implement an autoencoder in TensorFlow.

16.2 Introduction

Autoencoders are a kind of neural networks which imitate their inputs and produce the exact information at their outputs. They usually include two parts: Encoder and Decoder. The encoder transforms the input into a hidden space (hidden layer). The decoder then reconstructs the input information as the output. There are various types of autoencoders:

- **Undercomplete Autoencoders:** In this type, the hidden dimension is smaller than the input dimension. Training such autoencoder lead to capturing the most prominent features. However, using an overparameterized architecture in case of a lack of sufficient training data create overfitting and bars learning valuable features. A linear decoder can operate as PCA. However, the existence of non-linear functions create a more powerful dimensionality reduction model.
- **Regularized Autoencoders:** Instead of limiting the dimension of an autoencoder and the hidden layer size for feature learning, a loss function will be added to prevent overfitting.
- **Sparse Autoencoders:** Sparse autoencoders allow for representing the information bottleneck without demanding a decrease in the size of the hidden layer. Instead, it operates based on a loss function that penalizes the activations inside a layer.
- **Denoising Autoencoders (DAE):** We want an autoencoder to be sufficiently sensitive to regenerate the original input but not strictly sensitive so the model can learn a generalizable encoding and decoding. The approach is to insignificantly corrupt the input data with some noise with an uncorrupted data as the target output..
- **Contractive Autoencoders (CAE):** In this type of autoencoders, for small input variations, the encoded features should also be very similar. Denoising autoencoders force the reconstruction function to resist minor changes of the input, while contractive autoencoders enforce the encoder to resist against the input perturbation.

- **Variational Autoencoders:** A variational autoencoder (VAE) presents a probabilistic fashion for explaining an observation in hidden space. Therefore, instead of creating an encoder which results in a value to represent each latent feature, the encoder produces a probability distribution for each hidden feature.

In this post, we are going to design an Undercomplete Autoencoder in TensorFlow to train a low dimension representation.

16.3 Create an Undercomplete Autoencoder

We are working on building an autoencoder with a 3-layer encoder and 3-layer decoder. Each layer of encoder compresses its input along the spatial dimensions by a factor of two. Similarly, each segment of the decoder increases its input dimensionality by a factor of two.

```
import tensorflow.contrib.layers as lays

def autoencoder(inputs):
    # encoder
    # 32 file code blockx 32 x 1   ->  16 x 16 x 32
    # 16 x 16 x 32  ->  8 x 8 x 16
    # 8 x 8 x 16    ->  2 x 2 x 8
    net = lays.conv2d(inputs, 32, [5, 5], stride=2, padding='SAME')
    net = lays.conv2d(net, 16, [5, 5], stride=2, padding='SAME')
    net = lays.conv2d(net, 8, [5, 5], stride=4, padding='SAME')
    # decoder
    # 2 x 2 x 8    ->  8 x 8 x 16
    # 8 x 8 x 16   ->  16 x 16 x 32
    # 16 x 16 x 32  ->  32 x 32 x 1
    net = lays.conv2d_transpose(net, 16, [5, 5], stride=4, padding='SAME')
    net = lays.conv2d_transpose(net, 32, [5, 5], stride=2, padding='SAME')
    net = lays.conv2d_transpose(net, 1, [5, 5], stride=2, padding='SAME', activation_
↪fn=tf.nn.tanh)
    return net
```

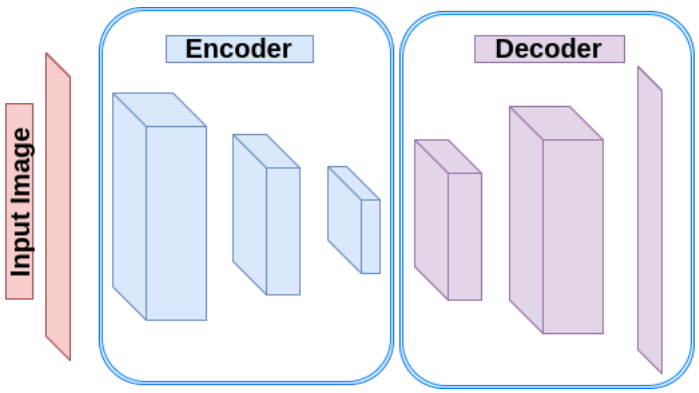


Fig. 1: **Figure 1:** Autoencoder

The MNIST dataset contains vectorized images of 28X28. Therefore we define a new function to reshape each batch of MNIST images to 28X28 and then resize to 32X32. The reason of resizing to 32X32 is to make it a power of two and therefore we can easily use the stride of 2 for downsampling and upsampling.

```
import numpy as np
from skimage import transform
```

(continues on next page)

(continued from previous page)

```
def resize_batch(imgs):
    # A function to resize a batch of MNIST images to (32, 32)
    # Args:
    #   imgs: a numpy array of size [batch_size, 28 X 28].
    # Returns:
    #   a numpy array of size [batch_size, 32, 32].
    imgs = imgs.reshape((-1, 28, 28, 1))
    resized_imgs = np.zeros((imgs.shape[0], 32, 32, 1))
    for i in range(imgs.shape[0]):
        resized_imgs[i, ..., 0] = transform.resize(imgs[i, ..., 0], (32, 32))
    return resized_imgs
```

Now we create an autoencoder, define a square error loss and an optimizer.

```
import tensorflow as tf

ae_inputs = tf.placeholder(tf.float32, (None, 32, 32, 1))  # input to the network␣
↪(MNIST images)
ae_outputs = autoencoder(ae_inputs)  # create the Autoencoder network

# calculate the loss and optimize the network
loss = tf.reduce_mean(tf.square(ae_outputs - ae_inputs))  # claculate the mean square␣
↪error loss
train_op = tf.train.AdamOptimizer(learning_rate=lr).minimize(loss)

# initialize the network
init = tf.global_variables_initializer()
```

Now we can read the batches, train the network and finally test the network by reconstructing a batch of test images.

```
from tensorflow.examples.tutorials.mnist import input_data

batch_size = 500  # Number of samples in each batch
epoch_num = 5     # Number of epochs to train the network
lr = 0.001        # Learning rate

# read MNIST dataset
mnist = input_data.read_data_sets("MNIST_data", one_hot=True)

# calculate the number of batches per epoch
batch_per_ep = mnist.train.num_examples // batch_size

with tf.Session() as sess:
    sess.run(init)
    for ep in range(epoch_num):  # epochs loop
        for batch_n in range(batch_per_ep):  # batches loop
            batch_img, batch_label = mnist.train.next_batch(batch_size)  # read a␣
↪batch
            batch_img = batch_img.reshape((-1, 28, 28, 1))               # reshape␣
↪each sample to an (28, 28) image
            batch_img = resize_batch(batch_img)                          # reshape␣
↪the images to (32, 32)
            _, c = sess.run([train_op, loss], feed_dict={ae_inputs: batch_img})
            print('Epoch: {} - cost= {:.5f}'.format((ep + 1), c))

    # test the trained network
```

(continues on next page)

(continued from previous page)

```
    batch_img, batch_label = mnist.test.next_batch(50)
    batch_img = resize_batch(batch_img)
    recon_img = sess.run([ae_outputs], feed_dict={ae_inputs: batch_img})[0]

    # plot the reconstructed images and their ground truths (inputs)
    plt.figure(1)
    plt.title('Reconstructed Images')
    for i in range(50):
        plt.subplot(5, 10, i+1)
        plt.imshow(recon_img[i, ..., 0], cmap='gray')
    plt.figure(2)
    plt.title('Input Images')
    for i in range(50):
        plt.subplot(5, 10, i+1)
        plt.imshow(batch_img[i, ..., 0], cmap='gray')
    plt.show()
```

CHAPTER 17

LICENSE