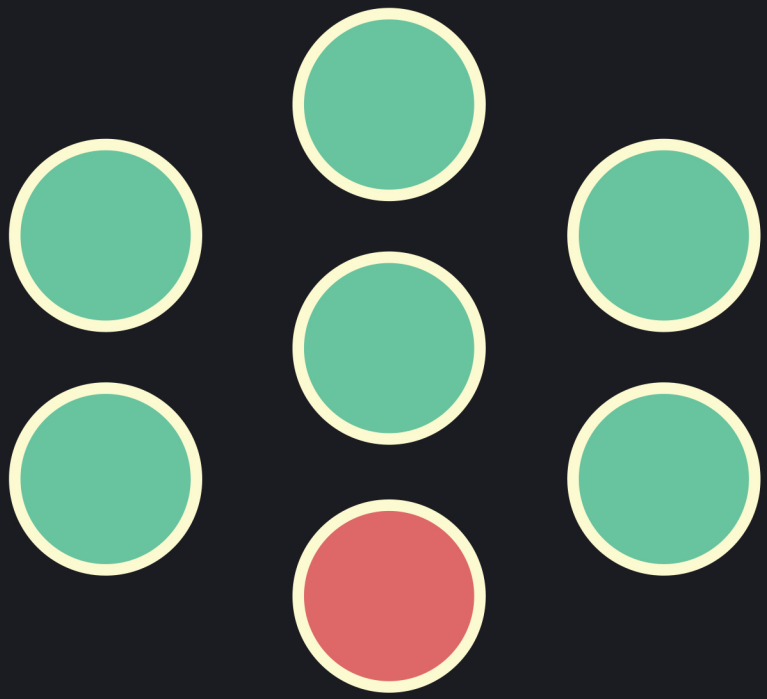
Venelin Valkov's

Hands-On Machine Learning from Scratch



Hands-On Machine Learning from Scratch

Develop a Deeper Understanding of Machine Learning Models by Implementing Them from Scratch in Python

Venelin Valkov

This book is for sale at http://leanpub.com/hmls

This version was published on 2019-06-29

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Venelin Valkov by spreading the word about this book on Twitter!

The suggested hashtag for this book is #hmls.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#hmls

Contents

Machine Learning fundamentals

> TL;DR Learn about the basics of Machine Learning - what types of learning exist, why implement algorithms from scratch and can you really trust your models?

Sick of being a lamer?

Here's a definition for "lamer" from urban dictionary:

> **Lamer** is someone who thinks they are smart, yet are really a loser. E.g. **that hacker wannabe is really just a lamer**.

You might've noticed that reading through Deep Learning and TensorFlow/PyTorch tutorials might give you an idea of how to do a specific task, but fall short if you want to apply it to your own problems. Importing a library and calling 4-5 methods might get the job done but leave you clueless about why it works. Can you solve this problem?

> "What I cannot create, I do not understand" - Richard Feynman

Different people learn in different styles, but all hackers learn the same way - we build stuff!

How this book helps?

- Provide a clear path to learning (reduce choice overload and paralysis by analysis) increasingly complex Machine Learning models
- Succinct implementations of Machine Learning algorithms solving real-world problems you can tinker with
- Just enough theory + math that helps you understand why things work after you understand what problem you have to solve

What is Machine Learning?

Machine Learning (ML) is the art and science of teaching machines to do complex tasks without explicitly programming them. Your job, as a hacker, is to:

- Define the problem in a way that a computer can understand it
- Choose a set of possible models that could solve it
- Feed the data
- Evaluate the performance and improve

There are 3 main types of learning: supervised, unsupervised and reinforcement

Supervised Learning

In *Supervised Learning* setting, you have a dataset, which is a collection of N labeled examples. Each example has a vector of features x_i and a label y_i. The label y_i can belong to a finite set of classes $\{1, \ldots, C\}$, real number or something more complex.

The goal of supervised learning algorithms is to build a model that receives a feature vector x as input and infer the correct label for it.

Unsupervised Learning

In *Unsupervised Learning* setting, you have a dataset of N **unlabeled examples**. Again, you have a feature vector x, and the goal is to build a model that takes it and transforms it into another. Some practical examples include **clustering**, **reducing the number of dimensions** and **anomaly detection**.

Reinforcement Learning

Reinforcement Learning is concerned with building agents that interact with an environment by getting its state and executing an action. Actions provide rewards and change the state of the environment. The goal is to learn a set of actions that maximize the total reward.

What Learning Algorithms are made of?

Each learning algorithm we're going to have a look at consists of three parts

- loss function - a measure of how wrong your model currently is
- optimization criteria based on the loss function
- optimization routine that uses data to find "good" solutions according to the optimization criteria

These are the main components of all the algorithms we're going to implement. While there are many optimization routines, Stochastic Gradient Descent[1] is the most used in practice. It is used to find optimal parameters for logistic regression, neural networks, and many other models.

Making predictions

How can you guarantee that your model will make correct predictions when deployed in production? Well, only suckers think that this is possible.

[1] https://en.wikipedia.org/wiki/Stochastic_gradient_descent

> "All models are wrong, but some are useful." - George Box

That said, there are ways to increase the prediction accuracy of your models. If the data used for training were selected randomly, independently of one another and following the same procedure for generating it, then, it is more likely your model to learn better. Still, for situations that are less likely to happen, your model will probably make errors.

Generally, the larger the data set, the better the predictions you can expect.

Tools of the trade

We're going to use a lot of libraries provided by different kind people, but the main ones are NumPy, Pandas and Matplotlib.

NumPy

NumPy[2] is the fundamental package for scientific computing with Python. It contains among other things:

- a powerful N-dimensional array object
- sophisticated (broadcasting) functions
- tools for integrating C/C++ and Fortran code
- useful linear algebra, Fourier transform, and random number capabilities

Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data. Arbitrary data-types can be defined. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases.

Here's a super simple walkthrough:

```
1  import numpy as np
2
3  a = np.array([1, 2, 3]) # 1D array
4  type(a)
```

```
1  numpy.ndarray
```

[2] http://www.numpy.org/

```
a.shape
```

```
(3,)
```

We've created a 1-dimensional array with 3 elements. You can get the first element of the array:

```
a[0]
```

```
1
```

and change it:

```
a[0] = 5

a
```

```
array([5, 2, 3])
```

Let's create a 2D array

```
b = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]) # 2D array

b.shape
```

```
(3, 3)
```

We have a 3x3 matrix. Let's select only the first 2 rows:

```
b[0:2]
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

Pandas

pandas[3] is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.

We're going to use pandas as a holder for our datasets. The primary entity we'll be working with is the `DataFrame`. We'll also do some transformations/preprocessing and visualizations.

Let's start by creating a `DataFrame`:

[3] https://pandas.pydata.org/

```
import pandas as pd

df =  pd.DataFrame(dict(
  customers=["Jill", "Jane", "Hanna"],
  payments=[120, 180, 90]
))

df
```

customers	payments
Jill	120
Jane	180
Hanna	90

You can check the size of the data frame:

```
df.shape
```

```
(3, 2)
```

We have 3 rows and 2 columns. You can use head()[4] to render a preview of the first five rows:

```
df.head()
```

customers	payments
Jill	120
Jane	180
Hanna	90

Let's check for missing values:

```
df.isnull()
```

customers	payments
False	False
False	False
False	False

You can apply functions such as sum()[5] to columns like `payments`:

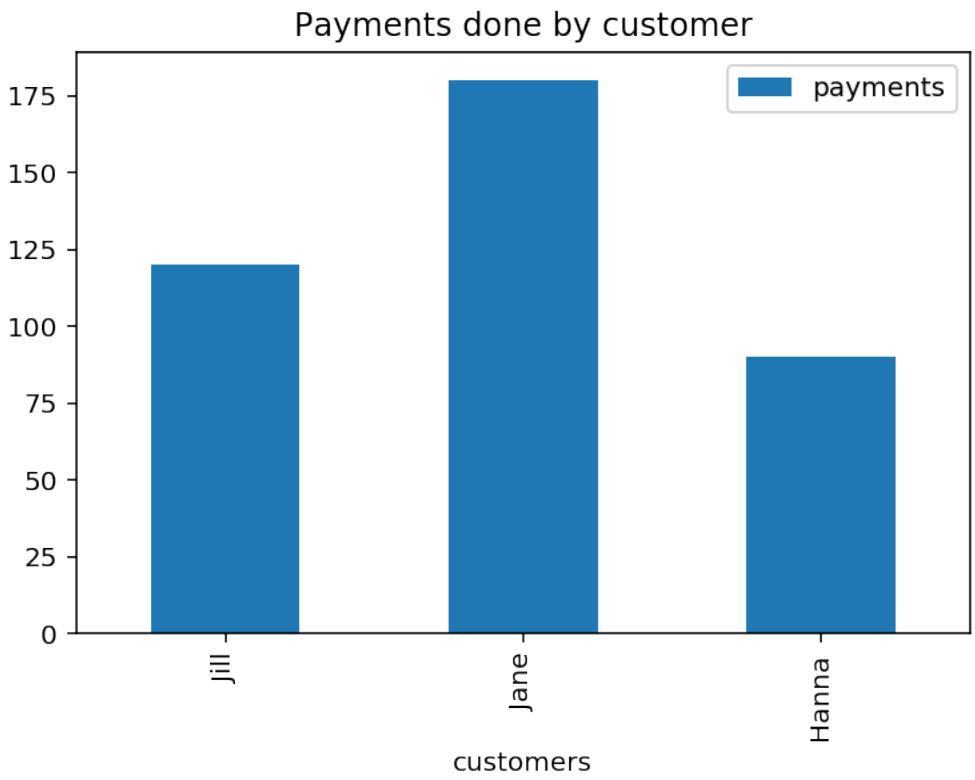
[4] https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.head.html
[5] https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.sum.html

```
df.payments.sum()
```

```
390
```

You can even show some charts using plot()[6]:

```
df.plot(
    kind='bar',
    x='customers',
    y='payments',
    title='Payments done by customer'
);
```



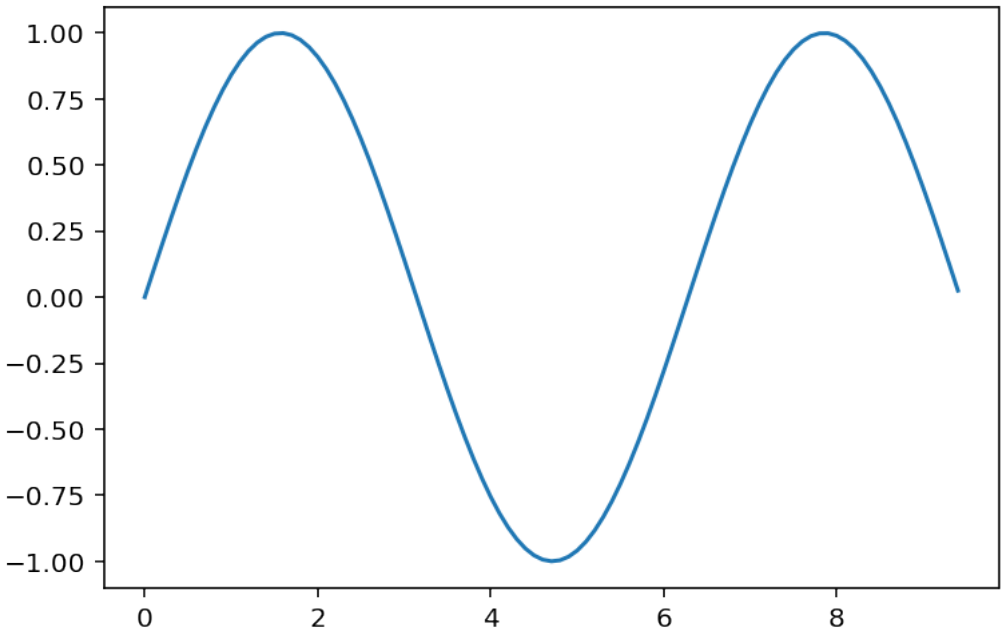
Matplotlib

> Matplotlib[7] is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms.

Here is a quick sample of it:

[6] https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.plot.html
[7] https://matplotlib.org/

```
import matplotlib.pyplot as plt

x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)

plt.plot(x, y);
```



Ready to start?

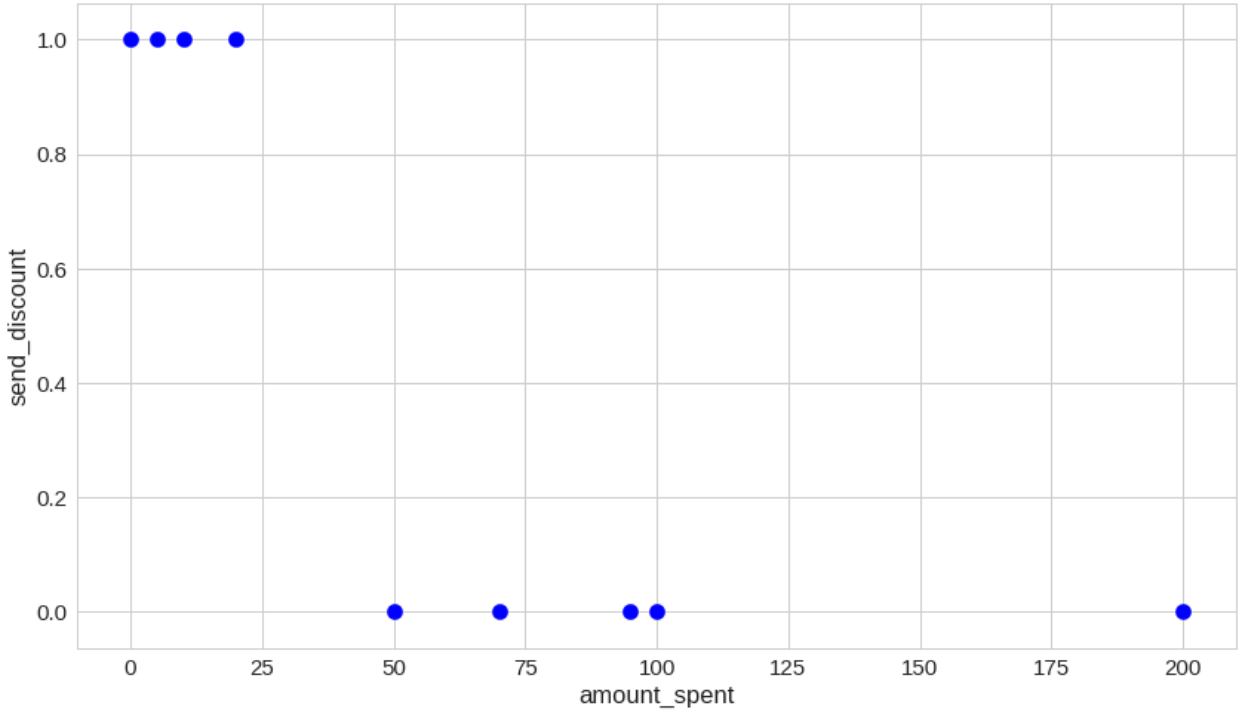
Welcome to the amazing world of Machine Learning. Let's get this party started!

Smart Discounts with Logistic Regression

Complete source code in Google Colaboratory Notebook[8]

The Data

You collected some data from your database(s), analytics packages, etc. Here's what you might've come up with:



Let's load the data into a Pandas data frame:

[8] https://colab.research.google.com/drive/1kmtjoULbyRtAtDPKYlhWSwATLpF7PQd8

```
data = OrderedDict(
        amount_spent =  [50,  10, 20, 5,  95,  70,  100,  200, 0],
        send_discount = [0,   1,  1,  1,  0,   0,   0,    0,   1]
    )
```

And have a look at it:

```
df = pd.DataFrame.from_dict(data)
```

	amount_spent	send_discount
0	50	0
1	10	1
2	20	1
3	5	1
4	95	0
5	70	0
6	100	0
7	200	0
8	0	1

> Note — the presented data is a simplification of a real dataset you might have. If your data is really simple, you might try simpler methods.

Making decisions with Logistic regression

Logistic regression is used for classification problems when the dependant/target variable is binary. That is, its values are true or false. *Logistic regression* is one of the most popular and widely used algorithms in practice (see this[9]).

Some problems that can be solved with Logistic regression include:

- **Email** — deciding if it is spam or not
- **Online transactions** — fraudulent or not
- **Tumor classification** — malignant or benign
- **Customer upgrade** — will the customer buy the premium upgrade or not

We want to predict the outcome of a variable y, such that:

$$y \in \{0, 1\}$$

and set *0*: negative class (e.g. email is not spam) or *1*: positive class (e.g. email is spam).

[9] https://www.kaggle.com/surveys/2017

Can‘t we just use Linear regression?

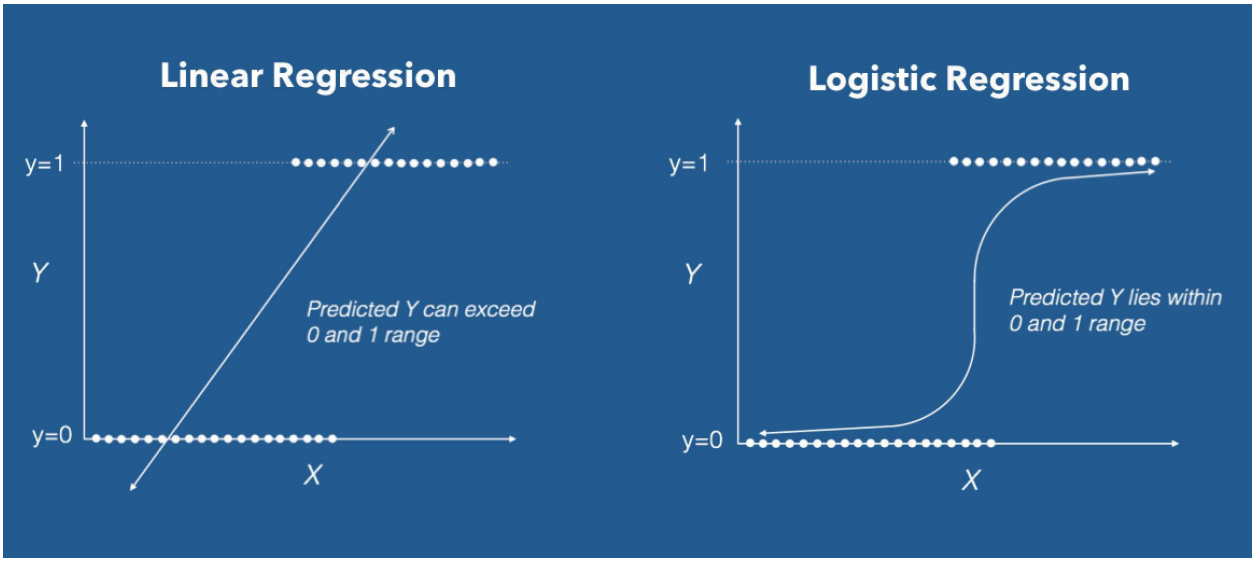


Image credit: machinelearningplus.com[10]

Linear regression is another very popular model. It works under the assumption that the observed phenomena (your data) are explainable by a straight line.

The response variable y of the Linear regression is *not restricted* within the $[0, 1]$ interval. That makes it pretty hard to take binary decisions based on its output. Thus, not suitable for our needs.

The Logistic Regression model

Given our problem, we want a model that uses 1 variable (predictor) (x_1 -amount_spent) to predict whether or not we should send a discount to the customer.

$$h_w(x) = w_1 x_1 + w_0$$

where the coefficients w_i are parameters of the model. Let the coefficient vector W be:

$$W = \begin{pmatrix} w_1 \\ w_0 \end{pmatrix}$$

Then we can represent $h_w(x)$ in a more compact form:

$$h_w(x) = w^T x$$

[10] https://machinelearningplus.com/

That is the *Linear Regression* model.

We want to build a model that outputs values that are between 0 and 1, so we want to come up with a hypothesis that satisfies:

$$0 \leq h_w(x) \leq 1$$

For ***Logistic Regression*** we want to modify this and introduce another function g:

$$h_w(x) = g(w^T x)$$

We're going to define ***g*** as:

$$g(z) = \frac{1}{1 + e^{-z}}$$

where

$$z \in \mathbb{R}$$

g is also known as the *sigmoid function* or the *logistic function*. After substitution, we end up with:

$$h_w(x) = \frac{1}{1 + e^{-(w^T x)}}$$

for our hypothesis.

A closer look at the sigmoid function

Intuitively, we're going to use the sigmoid function "over the" Linear regression model to bound it within [0;+1].

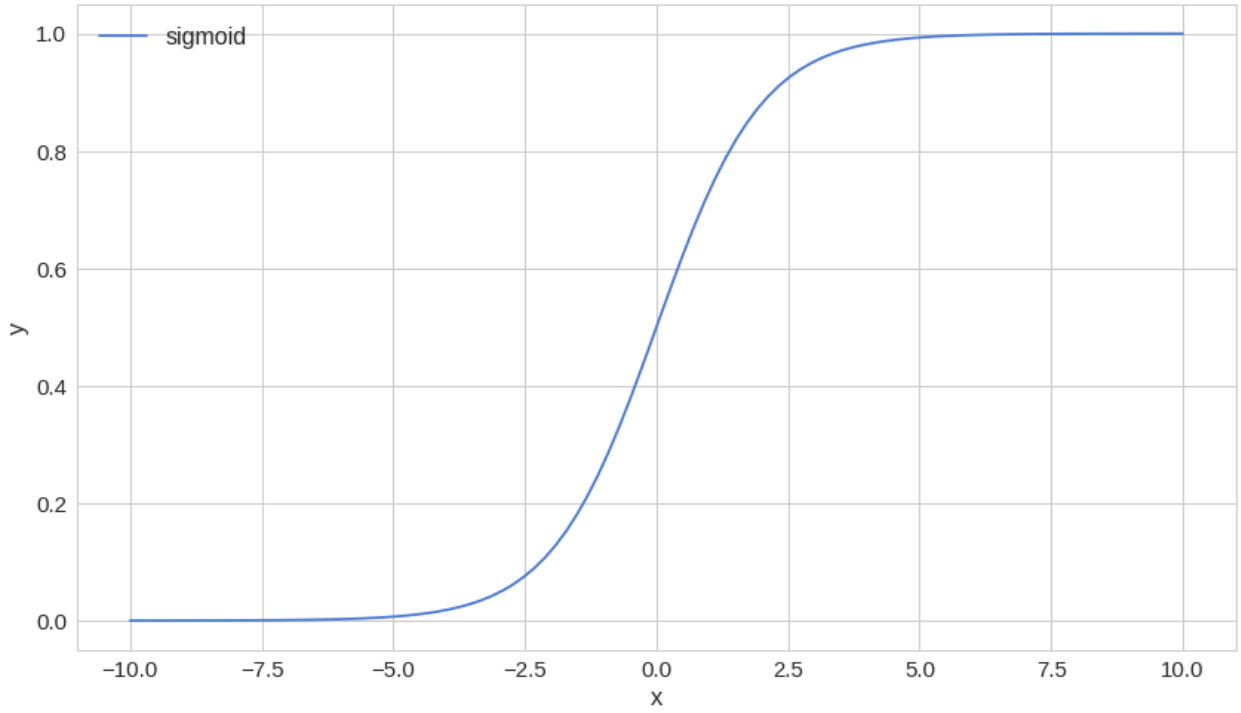
Recall that the sigmoid function is defined as:

$$g(z) = \frac{1}{1 + e^{-z}}$$

Let's translate it to a Python function:

```
1 def sigmoid(z):
2     return 1 / (1 + np.exp(-z))
```

A graphical representation of the sigmoid function:



It looks familiar, right? Notice how fast it converges to -1 or +1.

How can we find the parameters for our model?

Let's examine some approaches to find good parameters for our model. But what does good mean in this context?

Loss function

We have a model that we can use to make decisions, but we still have to find the parameters **W**. To do that, we need an objective measurement of how good a given set of parameters are. For that purpose, we will use a loss (cost) function:

$$J(W) = \frac{1}{m}\sum_{i=1}^{m} Cost(h_w(x^{(i)}), y^{(i)})$$

$$Cost(h_w(x), y) = \begin{cases} -log(h_w(x)) & \text{if} y = 1 \\ -log(1 - h_w(x)) & \text{if} y = 0 \end{cases}$$

Which is also known as the Log loss or Cross-entropy loss[11] function

[11] https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html

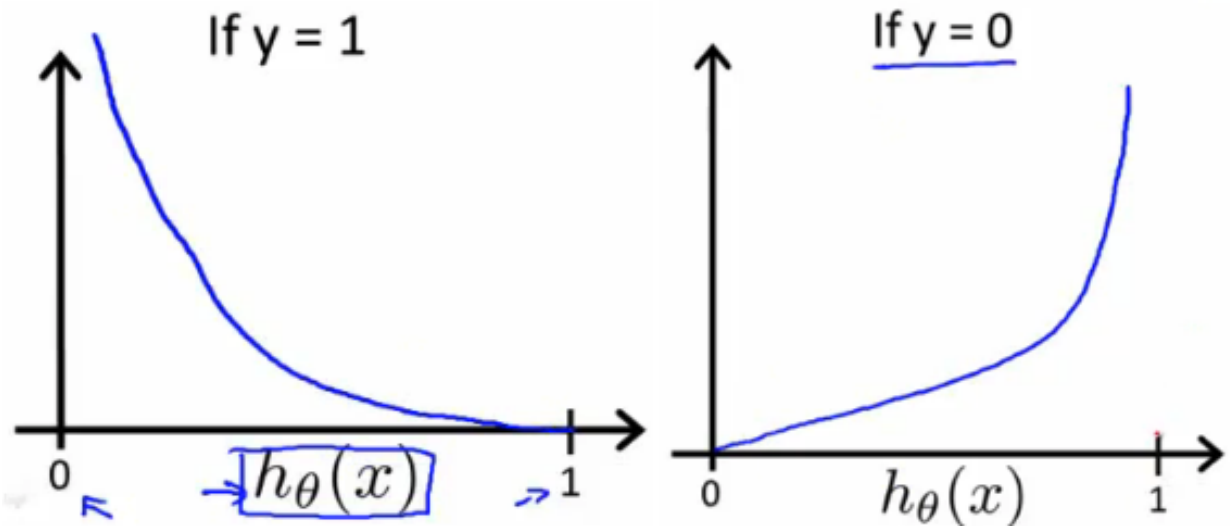


Image credit: ml-cheatsheet[12]

We can compress the above function into one:

$$J(W) = \frac{1}{m}(-y \log(h_w) - (1-y) \log(1-h_w))$$

where

$$h_w(x) = g(w^T x)$$

Let's implement it in Python:

```
def loss(h, y):
  return (-y * np.log(h) - (1 - y) * np.log(1 - h)).mean()
```

Approach #1 — tryout a number

Let's think of 3 numbers that represent the coefficients w_0, w_1, w_2.

[12] https://ml-cheatsheet.readthedocs.io

```
X = df['amount_spent'].astype('float').values
y = df['send_discount'].astype('float').values

def predict(x, w):
  return sigmoid(x * w)

def print_result(y_hat, y):
  print(f'loss: {np.round(loss(y_hat, y), 5)} predicted: {y_hat} actual: {y}')

y_hat = predict(x=X[0], w=.5)
print_result(y_hat, y[0])
```

```
loss: 25.0 predicted: 0.999999999986112 actual: 0.0
```

Unfortunately, I am pretty lazy, and this approach seems like a bit too much work for me. Let's go to the next one:

Approach #2 — tryout a lot of numbers

Alright, these days computers are pretty fast, 6+ core laptops are everywhere. Smartphones can be pretty performant, too! Let's use that power for good™ and try to find those pesky parameters by just trying out a lot of numbers:

```
for w in np.arange(-1, 1, 0.1):
  y_hat = predict(x=X[0], w=w)
  print(loss(y_hat, y[0]))
```

```
0.0
0.0
0.0
6.661338147750941e-16
9.359180097590508e-14
1.3887890837434982e-11
2.0611535832696244e-09
3.059022736706331e-07
4.539889921682063e-05
0.006715348489118056
0.6931471805599397
5.006715348489103
10.000045398900186
15.000000305680194
19.999999966169824
```

```
24.99999582410784
30.001020555434774
34.945041100449046
inf
inf
```

Amazing, the first parameter value we tried got us a loss of 0. Is it your lucky day or this will always be the case, though? The answer is left as an exercise for the reader :)

Approach #3 — Gradient descent

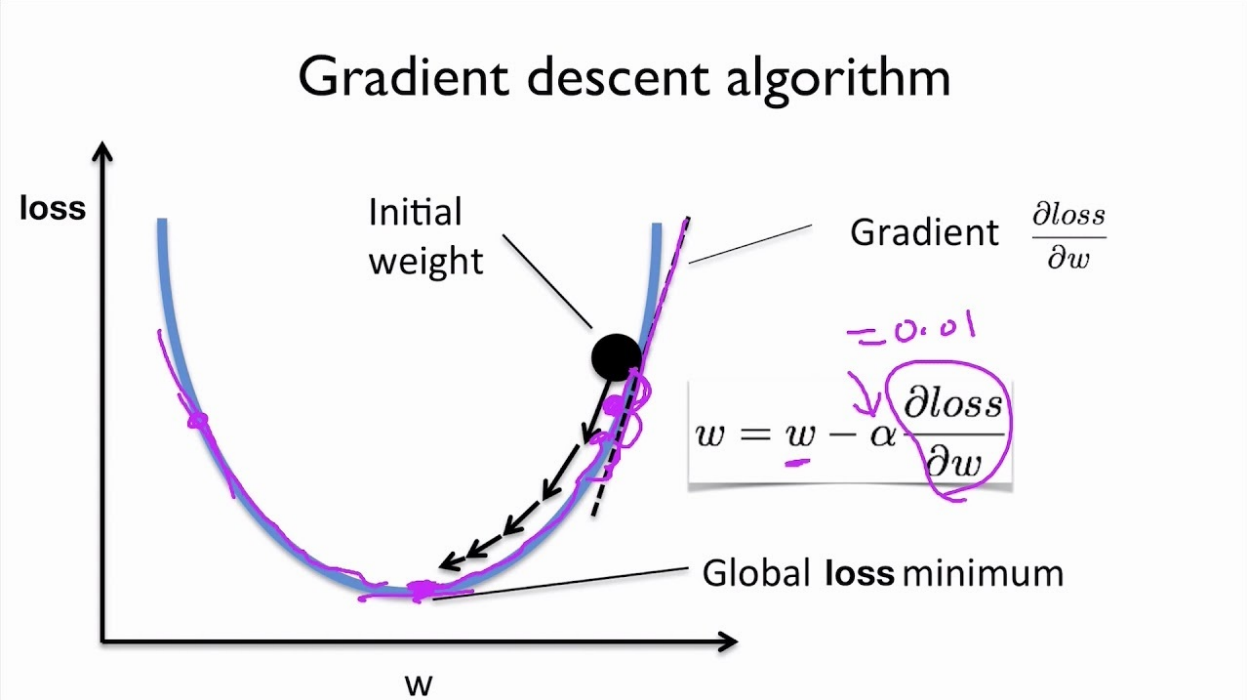


Image credit: PyTorchZeroToAll[13]

Gradient descent algorithms (yes, there are a lot of them) provide us with a way to find a minimum of some function f. They work by iteratively going in the direction of the descent as defined by the gradient.

In Machine Learning, we use gradient descent algorithms to find "good" parameters for our models (Logistic Regression, Linear Regression, Neural Networks, etc…).

How does it work? Starting somewhere, we take our first step downhill in the direction specified by the negative gradient. Next, we recalculate the negative gradient and take another step in the direction it specifies. This process continues until we get to a point where we can no longer move downhill — a local minimum.

[13] https://github.com/hunkim/PyTorchZeroToAll

Ok, but how can we find that gradient thing? We have to find the derivate of our cost function since our example is rather simple.

The first derivative of the sigmoid function

The first derivative of the sigmoid function is given by the following equation:

$$g'(z) = g(z)(1 - g(z))$$

Complete derivation can be found here[14].

The first derivative of the cost function

Recall that the cost function was given by the following equation:

$$J(W) = \frac{1}{m}(-y \log(h_w) - (1 - y) \log(1 - h_w))$$

Given

$$g'(z) = g(z)(1 - g(z))$$

We obtain the first derivative of the cost function:

$$\frac{\partial J(W)}{\partial W} = \frac{1}{m}(y(1 - h_w) - (1 - y)h_w)x = \frac{1}{m}(y - h_w)x$$

Updating our parameters W

Now that we have the derivate, we can go back to our updating rule and use it there:

$$W := W - \alpha(\frac{1}{m}(y - h_w)x)$$

The parameter a is known as *learning rate*. High learning rate can converge quickly, but risks overshooting the lowest point. Low learning rate allows for confident moves in the direction of the negative gradient. However, it time-consuming so it will take us a lot of time to get to converge.

[14] https://math.stackexchange.com/a/1225116/499458

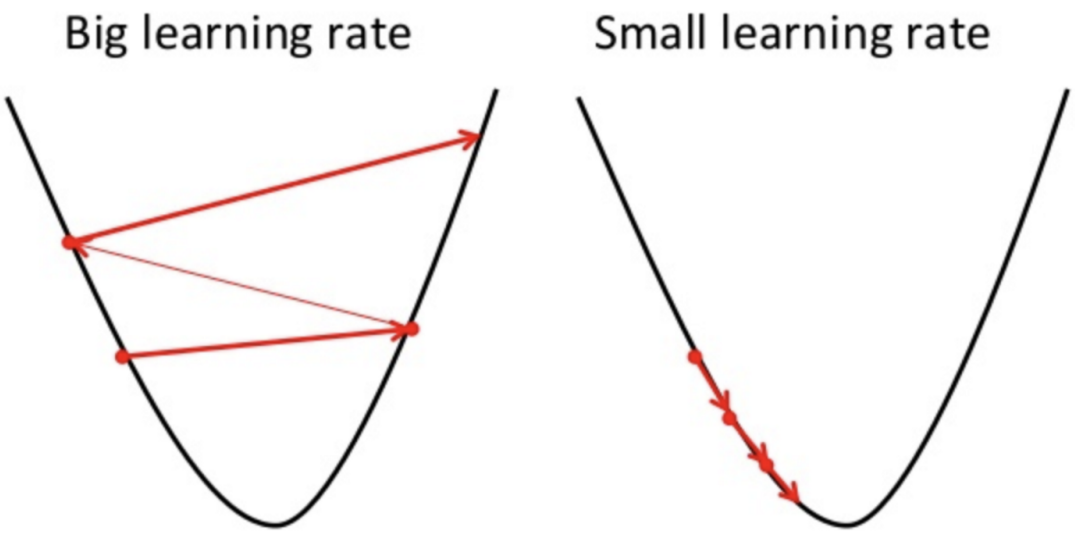


Image credit: Towards Data Science[15]

The Gradient descent algorithm

The algorithm we're going to use works as follows:

```
Repeat until convergence {
  1. Calculate gradient average
  2. Multiply by learning rate
  3. Subtract from weights
}
```

Let's do this in Python:

```
def predict(X, W):
  return sigmoid(np.dot(X, W))

def fit(X, y, n_iter=100000, lr=0.01):

  W = np.zeros(X.shape[1])

  for i in range(n_iter):
      z = np.dot(X, W)
      h = sigmoid(z)
      gradient = np.dot(X.T, (h - y)) / y.size
```

[15] https://towardsdatascience.com/

```
12        W -= lr * gradient
13    return W
```

About that until convergence part. You might notice that we kinda brute-force our way around it. That is, we will run the algorithm for a preset amount of iterations. Another interesting point is the initialization of our weights W — initially set at zero.

Let's put our implementation to the test, literally. But first, we need a function that helps us predict y given some data X (predict whether or not we should send a discount to a customer based on its spending):

```
def predict(X, W):
  return sigmoid(np.dot(X, W))
```

Now for our simple test:

```
class TestGradientDescent(unittest.TestCase):

    def test_correct_prediction(self):
      global X
      global y
      if len(X.shape) != 2:
        X = X.reshape(X.shape[0], 1)
      w = fit(X, y)
      y_hat = predict(X, w).round()
      self.assertTrue((y_hat == y).all())
```

Note that we use reshape to add a dummy dimension to X. Further, after our call to predict, we round the results. Recall that the sigmoid function spits out (kinda like a dragon with an upset stomach) numbers in the [0; 1] range. We're just going to round the result in order to obtain our 0 or 1 (yes or no) answers.

run_tests()

Here is the result of running our test case:

F

Well, that's not good, after all that hustling we're nowhere near achieving our goal of finding good parameters for our model. But, what went wrong?

Welcome to your first model debugging session! Let's start by finding whether our algorithm improves over time. We can use our loss metric for that:

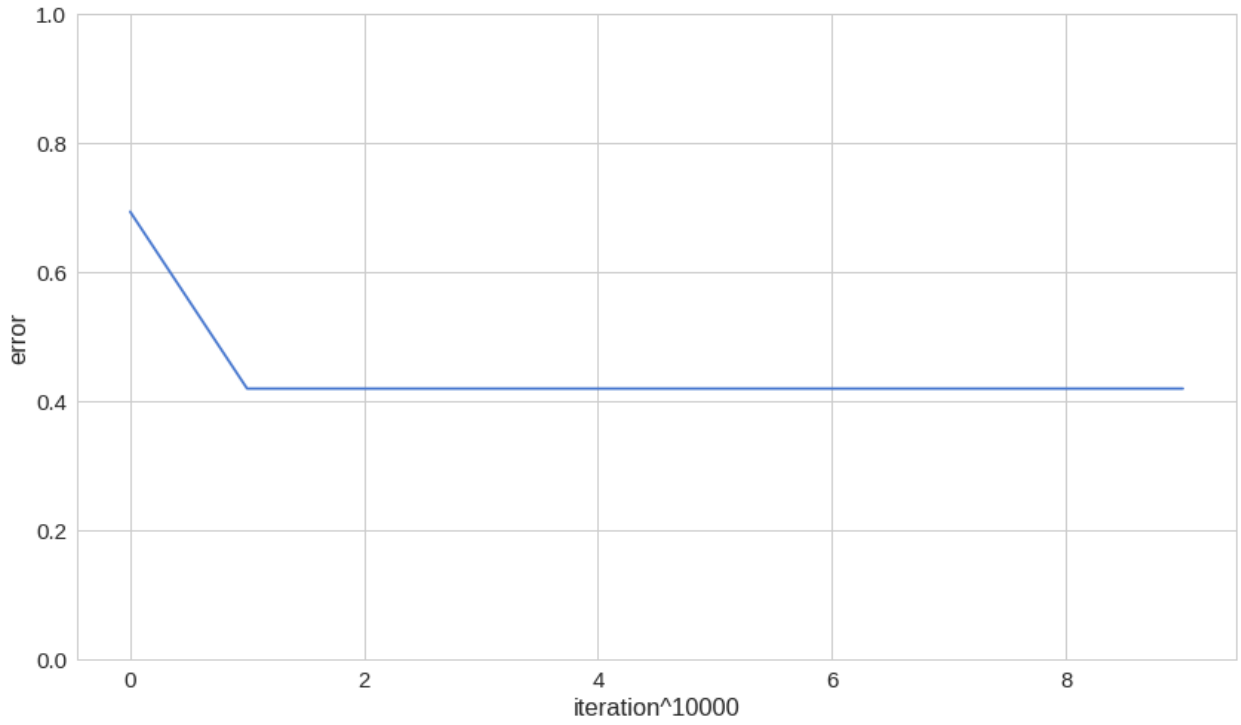
```
def fit(X, y, n_iter=100000, lr=0.01):

  W = np.zeros(X.shape[1])

  for i in range(n_iter):
      z = np.dot(X, W)
      h = sigmoid(z)
      gradient = np.dot(X.T, (h - y)) / y.size
      W -= lr * gradient

      if(i % 10000 == 0):
          e = loss(h, y)
          print(f'loss: {e} \t')

  return W
```

run_tests()

We pretty much copy & pasted our training code except that we're printing the training loss every *10,000* iterations. Let's have a look:

```
    loss: 0.6931471805599453
    loss: 0.41899283818630056
    loss: 0.41899283818630056
    loss: 0.41899283818630056
    loss: 0.41899283818630056
    loss: 0.41899283818630056
    loss: 0.41899283818630056
    loss: 0.41899283818630056
    loss: 0.41899283818630056
    loss: 0.41899283818630056
```

F.....



Suspiciously enough, we found a possible cause for our problem on the first try! Our loss doesn't get low enough, in other words, our algorithm gets stuck at some point that is not a good enough minimum for us. How can we fix this? Perhaps, try out different learning rate or initializing our parameter with a different value?

First, a smaller learning rate a :

```
def fit(X, y, n_iter=100000, lr=0.001):

  W = np.zeros(X.shape[1])

  for i in range(n_iter):
      z = np.dot(X, W)
      h = sigmoid(z)
      gradient = np.dot(X.T, (h - y)) / y.size
      W -= lr * gradient

      if(i % 10000 == 0):
        e = loss(h, y)
        print(f'loss: {e} \t')

  return W
```

```
run_tests()
```

With a=0.001 we obtain this:

```
loss: 0.42351356323845546
loss: 0.41899283818630056
loss: 0.41899283818630056
loss: 0.41899283818630056
loss: 0.41899283818630056
loss: 0.41899283818630056
loss: 0.41899283818630056
loss: 0.41899283818630056
loss: 0.41899283818630056
loss: 0.41899283818630056

F.......
```

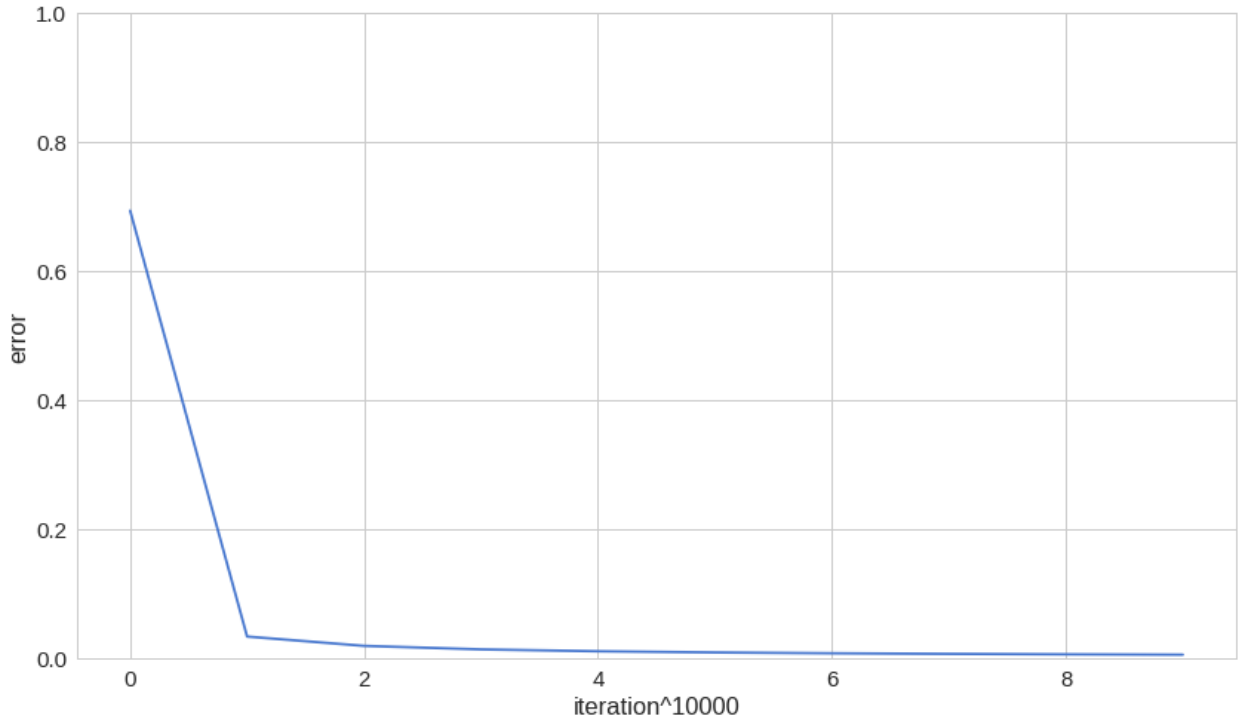
Not so successful, are we? How about adding one more parameter for our model to find/learn?

```
def add_intercept(X):
  intercept = np.ones((X.shape[0], 1))
  return np.concatenate((intercept, X), axis=1)

def predict(X, W):
  X = add_intercept(X)
  return sigmoid(np.dot(X, W))

def fit(X, y, n_iter=100000, lr=0.01):

  X = add_intercept(X)
  W = np.zeros(X.shape[1])

  for i in range(n_iter):
      z = np.dot(X, W)
      h = sigmoid(z)
      gradient = np.dot(X.T, (h - y)) / y.size
      W -= lr * gradient
  return W
```

```
run_tests()
```

And for the results:

```
........
----------------------------------------------------------------
Ran 8 tests in 0.686s

OK
```



What we did here? We added a new element to our parameter vector W and set it's initial value to 1. Seems like this turn things into our favor!

Bonus — building your own LogisticRegressor

Knowing all of the details of the inner workings of the Gradient descent is good, but when solving problems in the wild, we might be hard pressed for time. In those situations, a simple & easy to use interface for fitting a Logistic Regression model might save us a lot of time. So, let's build one!

But first, let's write some tests:

```
class TestLogisticRegressor(unittest.TestCase):

    def test_correct_prediction(self):
      global X
      global y
      X = X.reshape(X.shape[0], 1)
      clf = LogisticRegressor()
      y_hat = clf.fit(X, y).predict(X)
      self.assertTrue((y_hat == y).all())
```

```
class LogisticRegressor:

  def _add_intercept(self, X):
    intercept = np.ones((X.shape[0], 1))
    return np.concatenate((intercept, X), axis=1)

  def predict_probs(self, X):
    X = self._add_intercept(X)
    return sigmoid(np.dot(X, self.W))

  def predict(self, X):
    return self.predict_probs(X).round()

  def fit(self, X, y, n_iter=100000, lr=0.01):

    X = self._add_intercept(X)
    self.W = np.zeros(X.shape[1])

    for i in range(n_iter):
        z = np.dot(X, self.W)
        h = sigmoid(z)
        gradient = np.dot(X.T, (h - y)) / y.size
        self.W -= lr * gradient
    return self
```

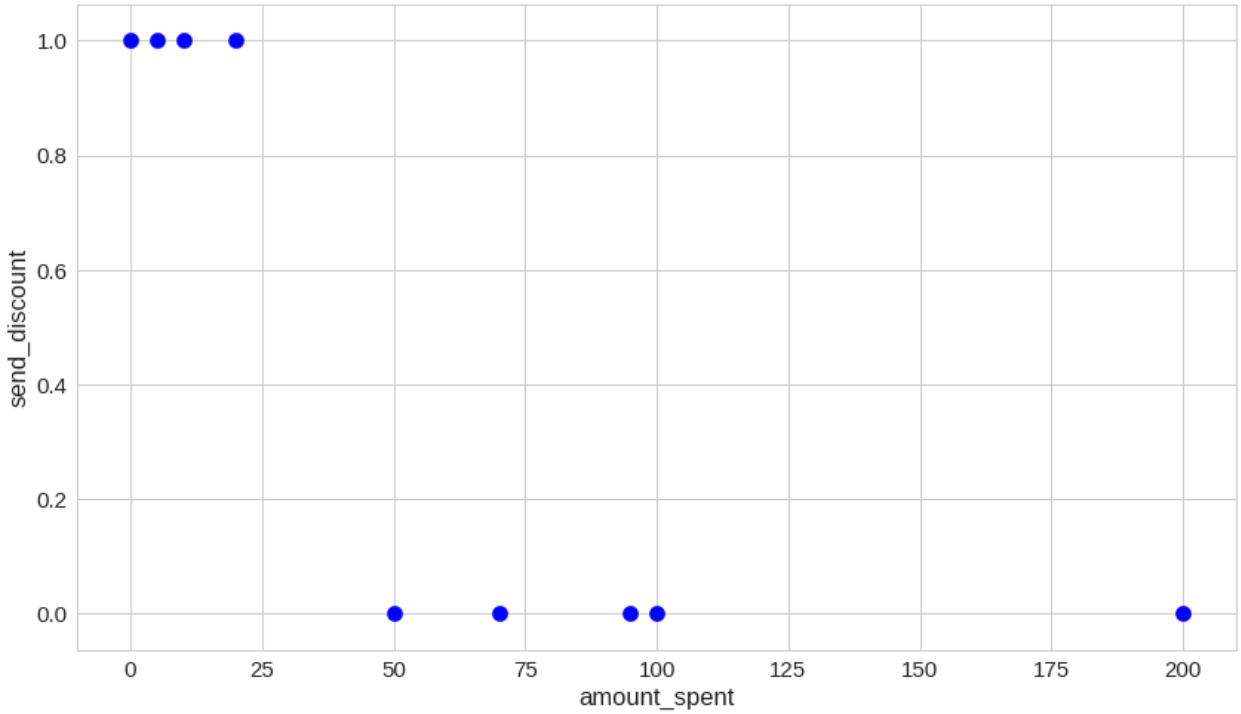
```
run_tests()
```

We just packed all previously written functions into a tiny class. One huge advantage of this approach is the fact that we hide the complexity of the Gradient descent algorithm and the use of the parameters $\boldsymbol{W}$.

Using our Regressor to decide who should receive discount codes

Now that you're done with the "hard" part let's use the model to predict whether or not we should send discount codes.

Let's recall our initial data:



Now let's try our model on data obtained from 2 new customers:

```
Customer 1 - $10
Customer 2 - $250
```

```
X_test = np.array([10, 250])
X_test = X_test.reshape(X_test.shape[0], 1)
y_test = LogisticRegressor().fit(X, y).predict(X_test)
```

```
y_test
```

Recall that 1 means send code and 0 means do not send:

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

Looks reasonable enough. Care to try out more cases?

Conclusion

Well done! You have a complete (albeit simple) LogisticRegressor implementation that you can play with. Go on, have some fun with it!

Complete source code in Google Colaboratory Notebook[16]

Coming up next, you will implement a Linear regression model from scratch :)

[16] https://colab.research.google.com/drive/1kmtjoULbyRtAtDPKYlhWSwATLpF7PQd8

Predicting House Prices with Linear Regression

> TL;DR Use a test-driven approach to build a Linear Regression model using Python from scratch. You will use your trained model to predict house sale prices and extend it to a multivariate Linear Regression.

I know that you've always dreamed of dominating the housing market. Until now, that was impossible. But with this limited offer you can... got a bit sidetracked there.

Let's start building our model with Python, but this time we will use it on a more realistic dataset.

Complete source code notebook[17]

The Data

Our data comes from a Kaggle competition named "House Prices: Advanced Regression Techniques[18]". It contains *1460* training data points and 80 features that might help us predict the selling price of a house.

Load the data

Let's load the Kaggle dataset into a Pandas data frame:

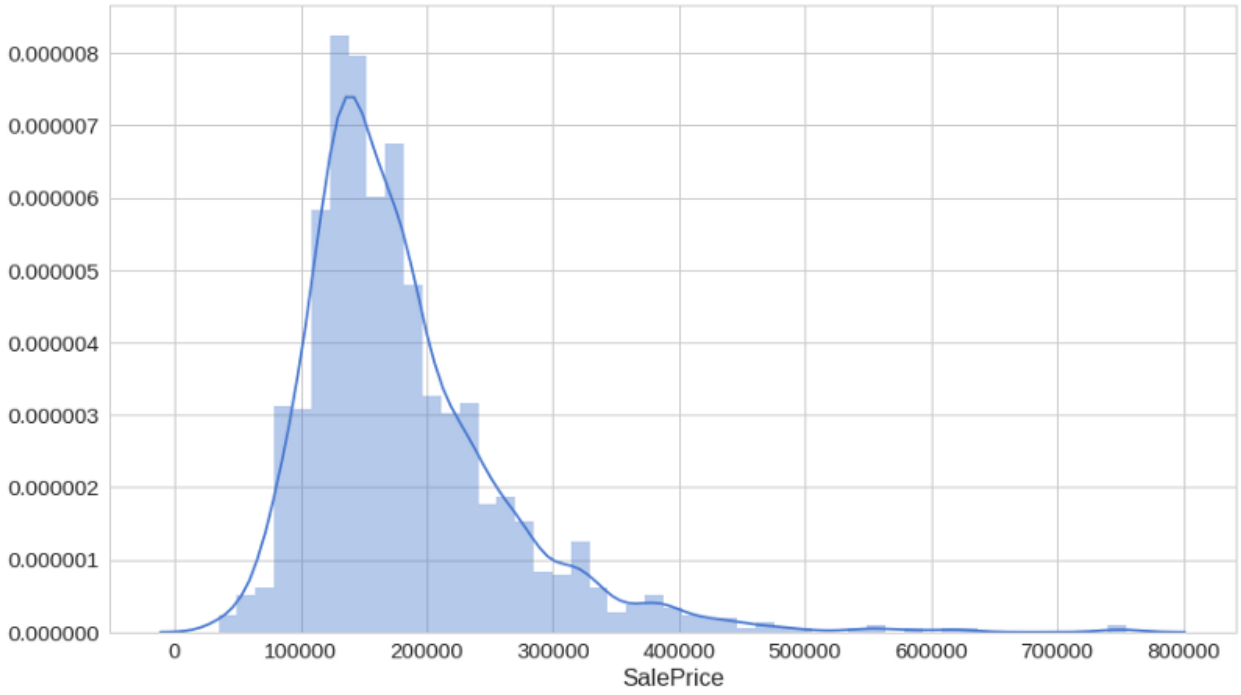
```
1 df_train = pd.read_csv('house_prices_train.csv')
```

Exploration — getting a feel for our data

We're going to predict the SalePrice column ($ USD), let's start with it:

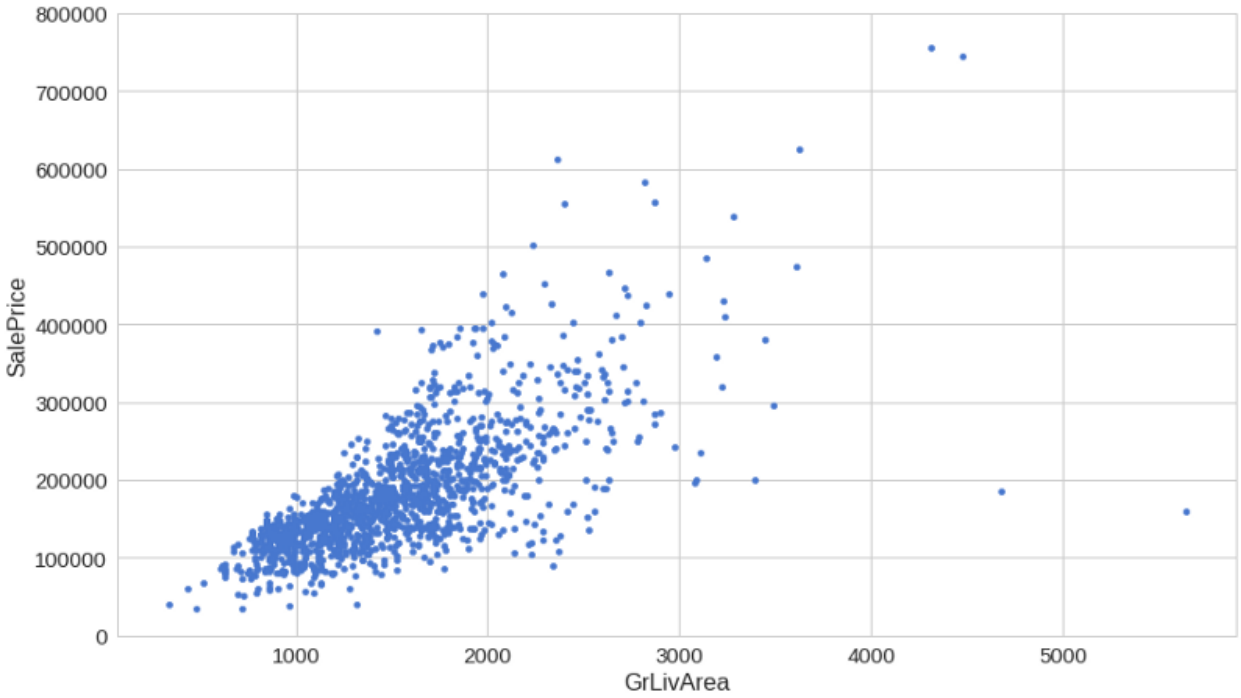
[17] https://colab.research.google.com/drive/1DXkpo9PmH9_HiCSz9NQlZ9vGQtMIYqmF
[18] https://www.kaggle.com/c/house-prices-advanced-regression-techniques

```
count      1460.000000
mean     180921.195890
std       79442.502883
min       34900.000000
25%      129975.000000
50%      163000.000000
75%      214000.000000
max      755000.000000
Name: SalePrice, dtype: float64
```



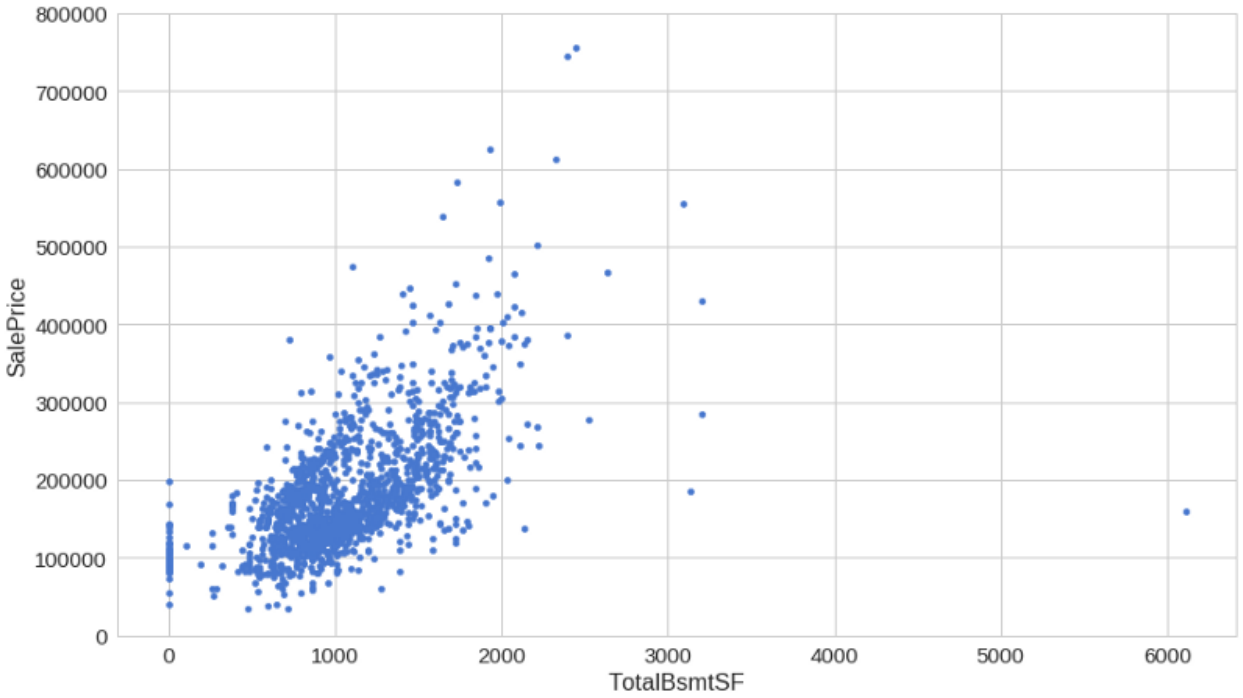
Most of the density lies between 100k and 250k, but there appears to be a lot of outliers on the pricier side.

Next, let's have a look at the greater living area (square feet) against the sale price:



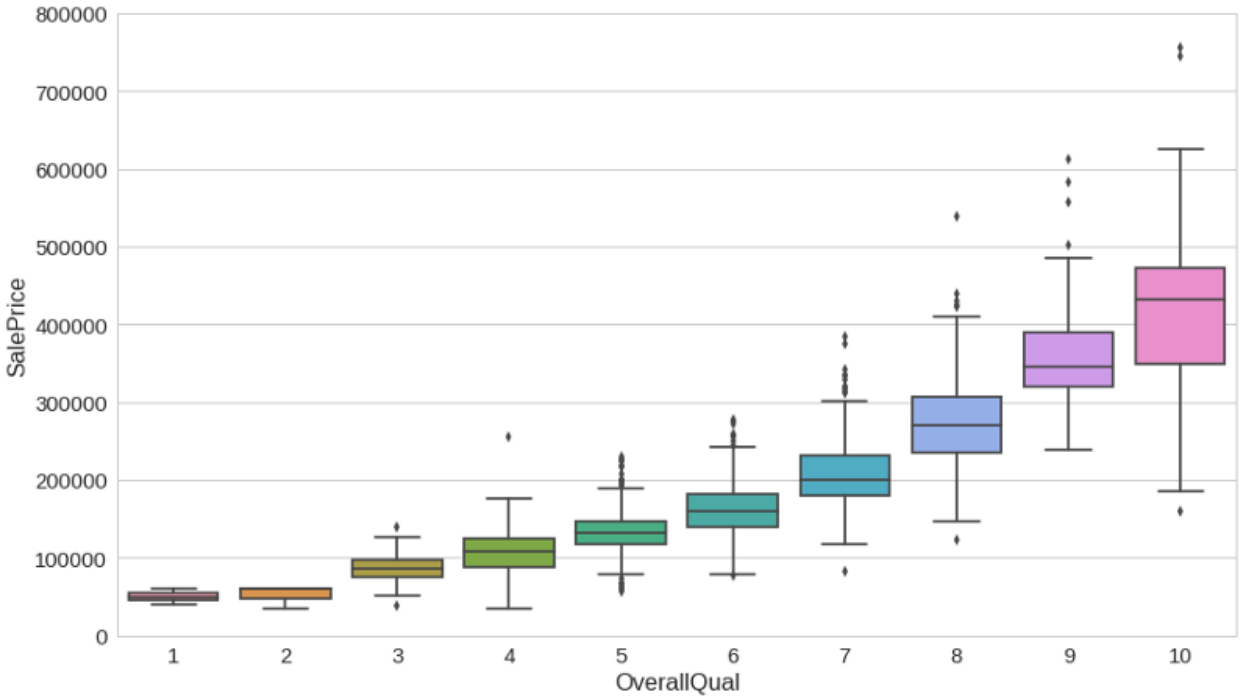
You might've expected that larger living area should mean a higher price. This chart shows you're generally correct. But what are those 2–3 "cheap" houses offering huge living area?

One column you might not think about exploring is the "TotalBsmtSF" — Total square feet of the basement area, but let's do it anyway:



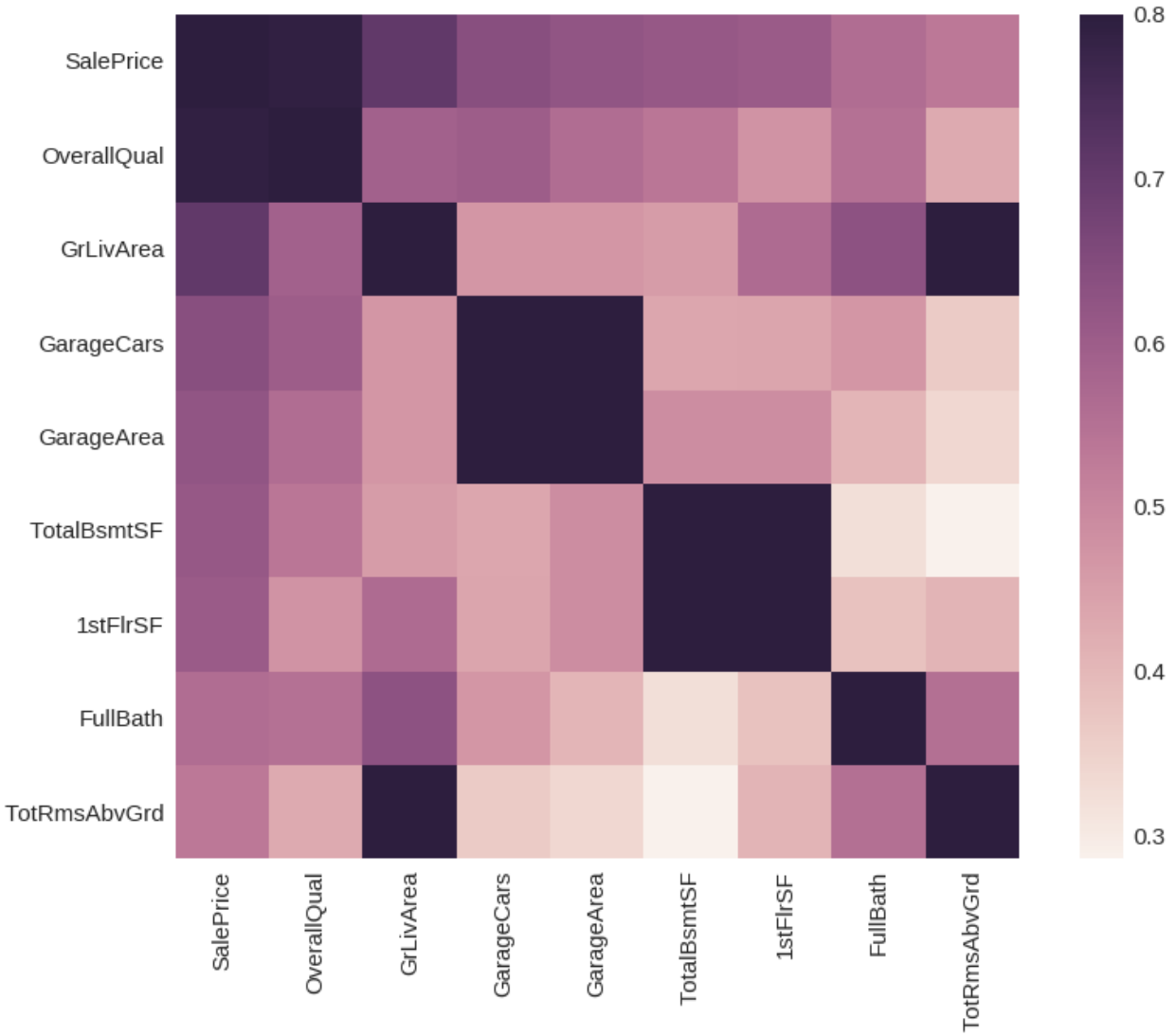
Intriguing, isn't it? The basement area seems like it might have a lot of predictive power for our model.

Ok, last one. Let's look at "OverallQual" — overall material and finish quality. Of course, this one seems like a much more subjective feature, so it might provide a bit different perspective on the sale price.



Everything seems fine for this one, except that when you look to the right things start getting much more nuanced. Will that "confuse" our model?

Let's have a more general view on the *top 8* correlated features with the sale price:



Surprised? All the features we discussed so far appear to be present. Its almost like we knew them from the start…

Do we have missing data?

We still haven't discussed ways to "handle" missing data, so we'll handle them like a boss — just not use those features:

	Row count	Percentage
PoolQC	1453	0.995205
MiscFeature	1406	0.963014
Alley	1369	0.937671
Fence	1179	0.807534
FireplaceQu	690	0.472603
LotFrontage	259	0.177397
GarageCond	81	0.055479
GarageType	81	0.055479
GarageYrBlt	81	0.055479
GarageFinish	81	0.055479
GarageQual	81	0.055479
BsmtExposure	38	0.026027
BsmtFinType2	38	0.026027
BsmtFinType1	37	0.025342
BsmtCond	37	0.025342

Yes, we're not going to use any of those.

Predicting the sale price

Now that we have some feel of the data we're playing with we can start our plan of attack — how to predict the sale price for a given house?

Using Linear Regression

Linear regression models assume that the relationship between a dependent continuous variable Y and one or more explanatory (independent) variables X is linear (that is, a straight line). It's used to predict values within a continuous range (e.g. sales, price) rather than trying to classify them into categories (e.g. cat, dog). Linear regression models can be divided into two main types:

Simple Linear Regression

Simple linear regression uses a traditional slope-intercept form, where a and b are the coefficients that we try to "learn" and produce the most accurate predictions. X represents our input data and Y is our prediction.

$$Y = bX + a$$

Multivariable Regression

A more complex, multi-variable linear equation might look like this, where w represents the coefficients or weights, our model will try to learn.

$$Y(x_1, x_2, x_3) = w_1 x_1 + w_2 x_2 + w_3 x_3 + w_0$$

The variables x_1, x_2, x_3 represent the attributes or distinct pieces of information, we have about each observation.

Loss function

Given our Simple Linear Regression equation:

$$Y = bX + a$$

We can use the following cost function to find the coefficients/parameters for our model:

Mean Squared Error (MSE) Cost Function

The MSE is defined as:

$$MSE = J(W) = \frac{1}{m} \sum_{i=1}^{m} (y^{(i)} - h_w(x^{(i)}))^2$$

where

$$h_w(x) = g(w^T x)$$

The MSE measures how much the average model predictions vary from the correct values. The number is higher when the model is performing "bad" on our training data.

The first derivative of MSE is given by:

$$MSE' = J'(W) = \frac{2}{m} \sum_{i=1}^{m} (h_w(x^{(i)}) - y^{(i)})$$

One Half Mean Squared Error (OHMSE)

We will apply a small modification to the MSE — multiply by *1/2* so when we take the derivative, the *2s* cancel out:

$$OHMSE = J(W) = \frac{1}{2m} \sum_{i=1}^{m} (y^{(i)} - h_w(x^{(i)}))^2$$

The first derivative of OHMSE is given by:

$$OHMSE' = J'(W) = \frac{1}{m}\sum_{i=1}^{m}(h_w(x^{(i)}) - y^{(i)})$$

Let's implement it in Python (yes, we're going TDD style!)

```
class TestLoss(unittest.TestCase):

  def test_zero_h_zero_y(self):
    self.assertAlmostEqual(loss(h=np.array([0]), y=np.array([0])), 0)

  def test_one_h_zero_y(self):
    self.assertAlmostEqual(loss(h=np.array([1]), y=np.array([0])), 0.5)

  def test_two_h_zero_y(self):
    self.assertAlmostEqual(loss(h=np.array([2]), y=np.array([0])), 2)

  def test_zero_h_one_y(self):
    self.assertAlmostEqual(loss(h=np.array([0]), y=np.array([1])), 0.5)

  def test_zero_h_two_y(self):
    self.assertAlmostEqual(loss(h=np.array([0]), y=np.array([2])), 2)
```

Now that we have the tests ready, we can implement the loss function:

```
def loss(h, y):
  sq_error = (h - y)**2
  n = len(y)
  return 1.0 / (2*n) * sq_error.sum()

run_tests()
```

time for the results:

```
.....
----------------------------------------------------------------------
Ran 5 tests in 0.007s

OK
```

Data preprocessing

We will do a little preprocessing to our data using the following formula (standardization):

$$x' = \frac{x - \mu}{\sigma}$$

where μ is the population mean and σ is the standard deviation.

But why? Why would we want to do that? The following chart might help you out:

What the doodle shows us that our old friend — the gradient descent algorithm, might converge (find good parameters) faster when our training data is scaled. Shall we?

```
x = df_train['GrLivArea']
y = df_train['SalePrice']

x = (x - x.mean()) / x.std()
x = np.c_[np.ones(x.shape[0]), x]
```

We will only use the greater living area feature for our first model.

Implementing Linear Regression

First, our tests:

```
class TestLinearRegression(unittest.TestCase):

    def test_find_coefficients(self):
      clf = LinearRegression()
      clf.fit(x, y, n_iter=2000, lr=0.01)
      np.testing.assert_array_almost_equal(
        clf._W,
        np.array([180921.19555322,  56294.90199925])
      )
```

Without further ado, the simple linear regression implementation:

```
class LinearRegression:

  def predict(self, X):
    return np.dot(X, self._W)

  def _gradient_descent_step(self, X, targets, lr):

    predictions = self.predict(X)

    error = predictions - targets
    gradient = np.dot(X.T,  error) / len(X)

    self._W -= lr * gradient

  def fit(self, X, y, n_iter=100000, lr=0.01):

    self._W = np.zeros(X.shape[1])

    for i in range(n_iter):
        self._gradient_descent_step(x, y, lr)

    return self
```

You might find our Linear Regression implementation simpler compared to the one presented for the Logistic Regression. Note that the use of the Gradient Descent algorithm is pretty much the same. Interesting, isn't it? A single algorithm can build two different types of models. Would we be able to use it for more?

```
run_tests()
```

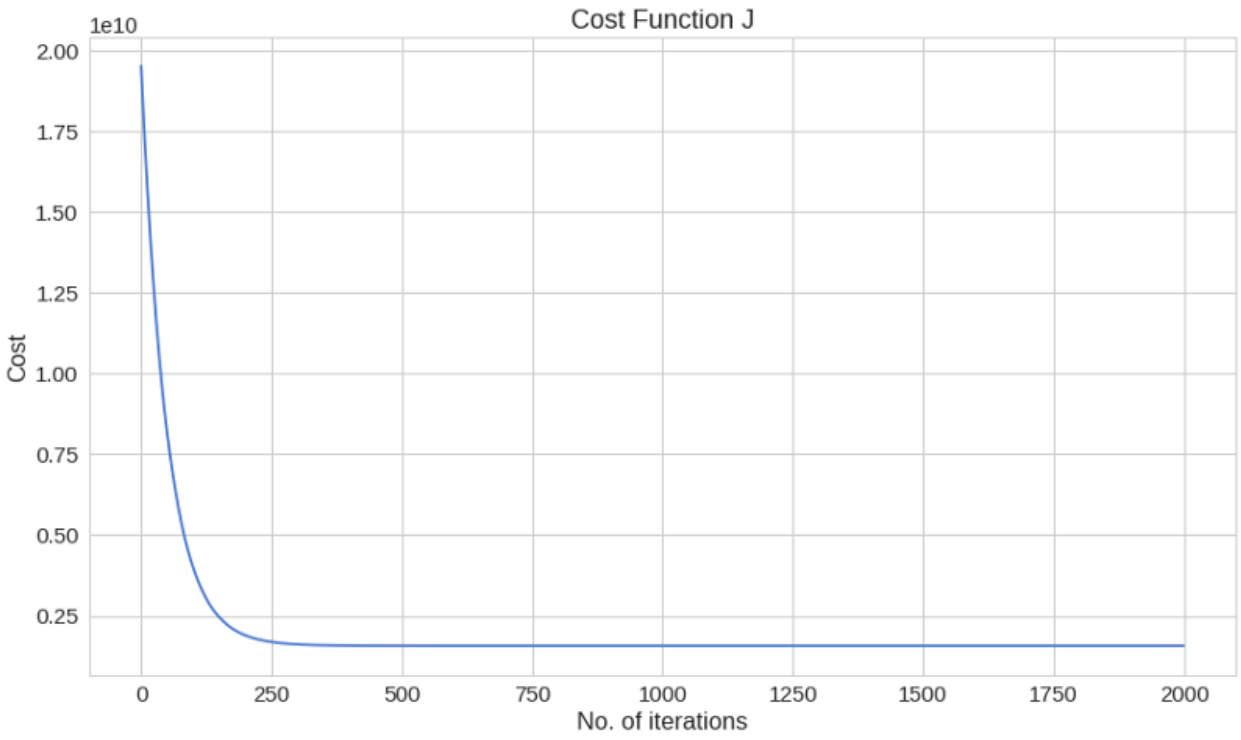
```
......
----------------------------------------------------------------------
Ran 6 tests in 1.094s

OK
```

Predicting the sale price with our first model

Let's use our freshly created model to begin our housing market domination:

```
clf = LinearRegression()
clf.fit(x, y, n_iter=2000, lr=0.01)
```

So how did the training go?



Our cost, at the last iteration, has a value of:

```
1569921604.8332634
```

Can we do better?

Multivariable Linear Regression

Let's use more of the available data to build a *Multivariable Linear Regression* model and see whether or not that will improve our OHMSE error. Let's not forget that scaling too:

```
x = df_train[['OverallQual', 'GrLivArea', 'GarageCars']]

x = (x - x.mean()) / x.std()
x = np.c_[np.ones(x.shape[0]), x]
```

Implementing *Multivariable Linear Regression*

This space is intentionally left (kinda) blank

Using Multivariable Linear Regression

Now that the implementation of our new model is done, we can use it. Done!?

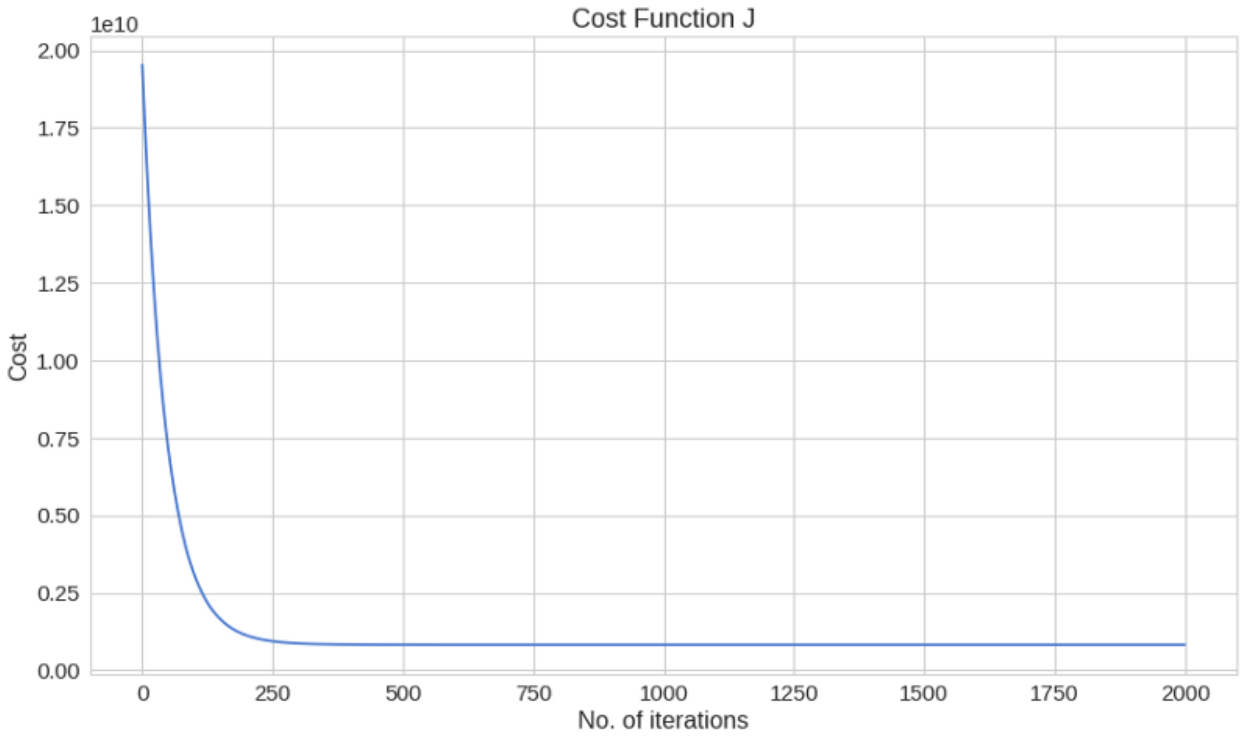
You see, young padawan, the magical world of software development has those mythical creates called **abstractions**. While it might be extraordinarily hard to get them right, they might reduce the complexity of the code you write by (pretty much) a ton.

You just used one such abstraction — called **vectorization**[19]. Essentially, that allowed us to build a *Multivariable Linear Regression* model without the need to loop over all features in our dataset. Neat, right?

Our client interface stays the same, too:

```
clf = LinearRegression()
clf.fit(x, y, n_iter=2000, lr=0.01)
```

And the results:




```
822817042.8437098
```

The loss at the last iteration is nearly 2 times smaller. Does this mean that our model is better now?

You can find complete source code and run the code in your browser[20]

[19] https://www.oreilly.com/library/view/python-for-data/9781449323592/ch04.html
[20] https://colab.research.google.com/drive/1DXkpo9PmH9_HiCSz9NQlZ9vGQtMIYqmF

Conclusion

Nice! You just implemented a Linear Regression model and not the simple/crappy kind.

One thing you might want to try is to predict house sale prices on the testing dataset from Kaggle. Is this model any good on it?

In the next part, you're going to implement a Decision Tree model from scratch!

Building a Decision Tree

Build a better house price prediction model using a Decision Tree

> TL;DR Build a Decision Tree regression model using Python from scratch. Compare the performance of your model with that of a Scikit-learn model. The Decision Tree is used to predict house sale prices and send the results to Kaggle.

I am sorry, you might be losing sleep. Deep down you know your Linear Regression model ain't gonna cut it. That housing market domination is still further down the road.

Can we improve that, can we have a model that makes better predictions?

Complete source code notebook[21]

The Data

Once again, we're going to use the Kaggle data: "House Prices: Advanced Regression Techniques[22]". It contains *1460* training data points and 80 features that might help us predict the selling price of a house.

Decision Trees

Decision tree models build structures like this:

[21] https://colab.research.google.com/drive/11T18ZELpOWuP7UtM7EKOut5juhJa9cHb
[22] https://www.kaggle.com/c/house-prices-advanced-regression-techniques

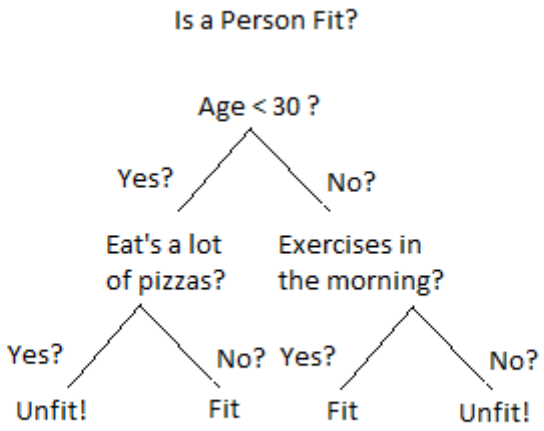


Image credit: xoriant.com[23]

The algorithms for building trees breaks down a data set into smaller and smaller subsets while an associated decision tree is incrementally developed. The final result is a tree with decision nodes and leaf nodes. A decision node has two or more branches. Leaf node represents a classification or decision (used for regression). The topmost decision node in a tree which corresponds to the best predictor (most important feature) is called a root node.

Decision trees can handle both categorical and numerical data. They are used for classification and regression problems. They can handle missing data pretty well, too!

Data Preprocessing

We're going to use the same data we used with the Linear Regression model. However, we're not going to do any scaling, just because we're lazy (or it is not needed):

```
1 X = df_train[['OverallQual', 'GrLivArea', 'GarageCars']]
2 y = df_train['SalePrice']
```

Cost function

We're going to use a new cost function — Root Mean Square Error (RMSE). It is the standard deviation of how far from the regression line data points are. In other words, it tells you how concentrated the data is around the line of best fit.

RMSE is given by the formula:

[23] https://www.xoriant.com/

$$RMSE = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(y^{(i)} - h(x^{(i)}))^2}$$

We've already implemented MSE in previous parts, so we're going to import an implementation here, in the name of readability (or holy laziness?):

```
from sklearn.metrics import mean_squared_error
from math import sqrt

def rmse(h, y):
  return sqrt(mean_squared_error(h, y))
```

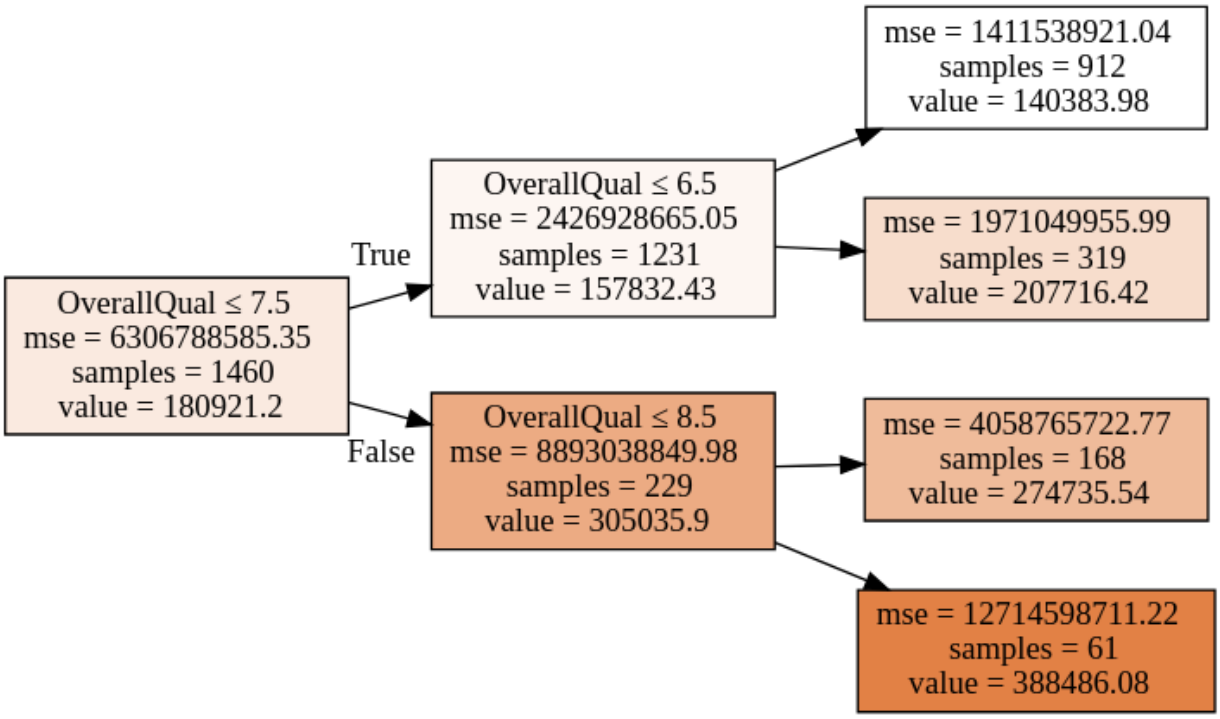
Using a prebuild Decision Tree model

Let use Decision Tree regressor from the scikit-learn[24] library to get a quick feel of the model:

```
from sklearn.ensemble import RandomForestRegressor

reg = RandomForestRegressor(
  n_estimators=1,
  max_depth=2,
  bootstrap=False,
  random_state=RANDOM_SEED
)
reg.fit(X, y)
```

We are using RandomForestRegressor with 1 estimator, which basically means we're using a Decision Tree model. Here is the tree structure of our model:

[24] https://scikit-learn.org/stable/



You should receive the exact same model (if you're running the code) since we are setting the random state. How many features did the model use?

Now that this model is ready to be used let's evaluate its R^2 score:

```
1 preds = reg.predict(X)
2 metrics.r2_score(y, preds)
```

```
0.6336246655552089
```

The R^2 statistic gives us information about the goodness of fit of the model. A score of 1 indicates perfect fit. Let's have a look at the RMSE:

```
48069.23940764968
```

Building your own Decision Tree

```
class DecisionTreeRegressor:

  def fit(self, X, y, min_leaf = 5):
    self.dtree = Node(X, y, np.array(np.arange(len(y))), min_leaf)
    return self

  def predict(self, X):
    return self.dtree.predict(X.values)
```

Let's start implementing our Node helper class:

```
class Node:

    def __init__(self, x, y, idxs, min_leaf=5):
        self.x = x
        self.y = y
        self.idxs = idxs
        self.min_leaf = min_leaf
        self.row_count = len(idxs)
        self.col_count = x.shape[1]
        self.val = np.mean(y[idxs])
        self.score = float('inf')
        self.find_varsplit()
```

Trees are recursive data structures and we're going to take full advantage of that. Our Node class represents one decision point in our model. Each division within the model has 2 possible outcomes for finding a solution — go to the left or go to the right. That decision point also divides our data into two sets.

The property idxs stores indexes of the subset of the data that this Node is working with.

The decision (prediction) is based on the value that Node holds. To make that prediction we're just going to take the average of the data of the dependent variable for this Node.

The method `find_varsplit` finds where should we split the data. Let's have a look at it:

```
def find_varsplit(self):
    for c in range(self.col_count): self.find_better_split(c)
    if self.is_leaf: return
    x = self.split_col
    lhs = np.nonzero(x <= self.split)[0]
    rhs = np.nonzero(x > self.split)[0]
    self.lhs = Node(self.x, self.y, self.idxs[lhs], self.min_leaf)
    self.rhs = Node(self.x, self.y, self.idxs[rhs], self.min_leaf)
```

First, we try to find a better feature to split on. If no such feature is found (we're at a leaf node) we do nothing. Then we use the split value found by find_better_split, create the data for the left and right nodes and create each one using a subset of the data.

Here are the `split_col` and `is_leaf` implementations:

```
@property
def split_col(self): return self.x.values[self.idxs,self.var_idx]

@property
def is_leaf(self): return self.score == float('inf')
```

It is time to implement the workhorse of our algorithm `find_better_split` method:

```
def find_better_split(self, var_idx):
    x = self.x.values[self.idxs, var_idx]

    for r in range(self.row_count):
        lhs = x <= x[r]
        rhs = x > x[r]
        if rhs.sum() < self.min_leaf or lhs.sum() < self.min_leaf: continue

        curr_score = self.find_score(lhs, rhs)
        if curr_score < self.score:
            self.var_idx = var_idx
            self.score = curr_score
            self.split = x[r]

def find_score(self, lhs, rhs):
    y = self.y[self.idxs]
    lhs_std = y[lhs].std()
    rhs_std = y[rhs].std()
    return lhs_std * lhs.sum() + rhs_std * rhs.sum()
```

We are trying to split on each data point and let the best split wins.

We're going to create our split such that it has as low standard deviation as possible. We find the split that minimizes the weighted averages of the standard deviations which is equivalent to minimizing RMSE.

If we find a better split we store the following information: index of the variable, split score and value of the split.

The score is a metric that tells us how effective the split was (note that leaf nodes do not have scores, so it will be infinity). The method find_score calculates a weighted average of the data. If the score is lower than the previous we have a better split. Note that the score is initially set to infinity -> only leaf nodes and really shallow trees (and Thanos) have a score of infinity.

Finally, let's look at how we use all this to make predictions:

```
def predict(self, x):
    return np.array([self.predict_row(xi) for xi in x])

def predict_row(self, xi):
    if self.is_leaf: return self.val
    node = self.lhs if xi[self.var_idx] <= self.split else self.rhs
    return node.predict_row(xi)
```

Once again, we're exploiting the recursive nature of life. Starting at the tree root, predict_row checks if we need to go left or right node based on the split value we found. The recursion ends once we hit a leaf node. At that point, the answer/prediction is stored in the val property.

Here is the complete source of our Node class:

```
class Node:

    def __init__(self, x, y, idxs, min_leaf=5):
        self.x = x
        self.y = y
        self.idxs = idxs
        self.min_leaf = min_leaf
        self.row_count = len(idxs)
        self.col_count = x.shape[1]
        self.val = np.mean(y[idxs])
        self.score = float('inf')
        self.find_varsplit()

    def find_varsplit(self):
        for c in range(self.col_count): self.find_better_split(c)
```

```
        if self.is_leaf: return
        x = self.split_col
        lhs = np.nonzero(x <= self.split)[0]
        rhs = np.nonzero(x > self.split)[0]
        self.lhs = Node(self.x, self.y, self.idxs[lhs], self.min_leaf)
        self.rhs = Node(self.x, self.y, self.idxs[rhs], self.min_leaf)

    def find_better_split(self, var_idx):

        x = self.x.values[self.idxs, var_idx]

        for r in range(self.row_count):
            lhs = x <= x[r]
            rhs = x > x[r]
            if rhs.sum() < self.min_leaf or lhs.sum() < self.min_leaf: continue

            curr_score = self.find_score(lhs, rhs)
            if curr_score < self.score:
                self.var_idx = var_idx
                self.score = curr_score
                self.split = x[r]

    def find_score(self, lhs, rhs):
        y = self.y[self.idxs]
        lhs_std = y[lhs].std()
        rhs_std = y[rhs].std()
        return lhs_std * lhs.sum() + rhs_std * rhs.sum()

    @property
    def split_col(self): return self.x.values[self.idxs,self.var_idx]

    @property
    def is_leaf(self): return self.score == float('inf')

    def predict(self, x):
        return np.array([self.predict_row(xi) for xi in x])

    def predict_row(self, xi):
        if self.is_leaf: return self.val
        node = self.lhs if xi[self.var_idx] <= self.split else self.rhs
        return node.predict_row(xi)
```

Evaluation

Let's check how your Decision Tree regressor does on the training data:

```
regressor = DecisionTreeRegressor().fit(X, y)
preds = regressor.predict(X)
```

Here is the R^2 score:

`0.8504381072711565`

Our *scikit-learn* model gave us a score of `0.6336246655552089`

The *RMSE* score is:

`30712.460628635836`

The *scikit-learn* regressor gave us `48069.23940764968`

Looks like your model is doing pretty good, eh? Let's make a prediction on the test data and send it to Kaggle.

Sending your predictions to Kaggle

Let make our predictions and format the data as requested on Kaggle[25]:

```
X_test = df_test[['OverallQual', 'GrLivArea', 'GarageCars']]
pred_test = regressor.predict(X_test)

submission = pd.DataFrame({'Id': df_test.Id, 'SalePrice': pred_test})
submission.to_csv('submission.csv', index=False)
```

Feel free to submit your csv file to Kaggle. Also, how can you improve?

Conclusion

Give yourself a treat, you just implemented a Decision Tree regressor!

Would it be possible to implement Random Forest regressor on top of your model? How that affects your Kaggle scores?

In the next part, you're gonna do some unsupervised learning with k means!

[25] https://www.kaggle.com/c/house-prices-advanced-regression-techniques#evaluation

Acknowledgments

The source code presented in this part is heavily inspired by the excellent course of fast.ai[26] — Introduction to Machine Learning for Coders[27]

[26] https://www.fast.ai/
[27] http://course18.fast.ai/ml

Color palette extraction with K-means clustering

Choosing a color palette for your next big mobile app (re)design can be a daunting task, especially when you don't know what the heck you're doing. How can you make it easier (asking for a friend)?

One approach is to head over to a place where the *PROs* share their work. Pages like Dribbble[28], uplabs[29] and Behance[30] have the goods.

After finding mockups you like, you might want to extract colors from them and use those. This might require opening specialized software, manually picking color with some tool(s) and other over-the-counter hacks. Let's use Machine Learning to make your life easier.

Complete source code notebook[31]

Unsupervised Learning

Up until now we only looked at models that require training data in the form of features and labels. In other words, for each example, we need to have the correct answer, too.

Usually, such training data is hard to obtain and requires many hours of manual work done by humans (yes, we're already serving "The Terminators"). Can we skip all that?

Yes, at least for some problems we can use example data without knowing the correct answer for it. One such problem is *clustering*.

What is clustering?

Given some vector of data points X, clustering methods allow you to put each point in a group. In other words, you can categorize a set of entities, based on their properties, automatically.

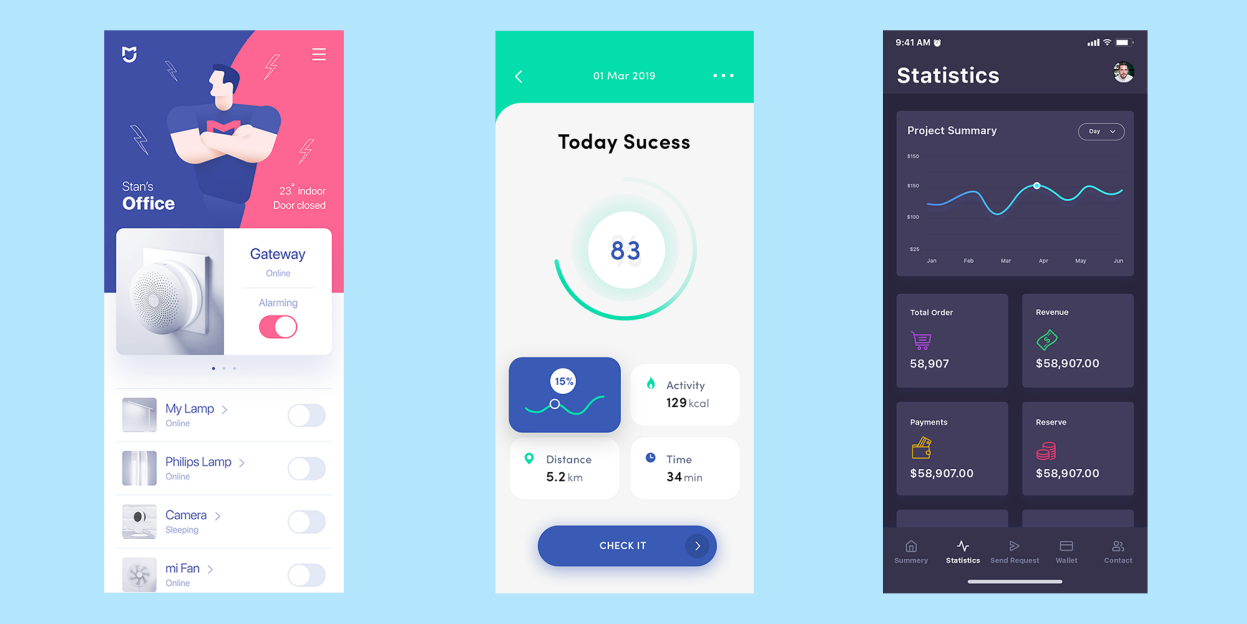
Yes, that is very useful in practice. Usually, you run the algorithm on a bunch of data points and specify how much groups you want. For example, your inbox contains two main groups of e-mail: spam and not-spam (were you waiting for something else?). You can let the clustering algorithm create two groups from your emails and use your beautiful brain to classify which is which.

More applications of clustering algorithms:

[28] https://dribbble.com/
[29] https://www.uplabs.com/
[30] https://www.behance.net/
[31] https://colab.research.google.com/drive/1_p1nptDfvJhcSvprmi1WtoIugsCI40Xa

- *Customer segmentation* - find groups of users that spend/behave the same way
- *Fraudulent transactions* - find bank transactions that belong to different clusters and identify them as fraudulent
- *Document analysis* - group similar documents

The Data



source: various authors on https://www.uplabs.com/

This time, our data doesn't come from some predefined or well-known dataset. Since Unsupervised learning does not require labeled data, the Internet can be your oyster.

Here, we'll use 3 mobile UI designs from various authors. Our model will run on each shot and try to extract the color palette for each.

What is K-Means Clustering?

K-Means Clustering is defined by Wikipedia[32] as:

> k-means clustering is a method of vector quantization, originally from signal processing, that is popular for cluster analysis in data mining. k-means clustering aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster. This results in a partitioning of the data space into Voronoi cells.

[32] https://en.wikipedia.org/wiki/K-means_clustering

Wikipedia also tells us that solving K-Means clustering is difficult (in fact, NP-hard[33]) but we can find local optimum solutions using some heuristics.

But how do *K-Means Clustering* works?

Let's say you have some vector X that contains n data points. Running our algorithm consists of the following steps:

1. Take random k points (called *centroids*) from X
2. Assign every point to the closest centroid. The newly formed bunch of points is called *cluster*.
3. For each cluster, find new centroid by calculating a new center from the points
4. Repeat steps 2-3 until centroids stop changing

Let's see how can we use it to extract color palettes from mobile UI screenshots.

Data Preprocessing

Given our data is stored in raw pixels (called images), we need a way to convert it to points that our clustering algorithm can use.

Let's first define two classes that represent a point and cluster:

```
class Point:

  def __init__(self, coordinates):
    self.coordinates = coordinates
```

Our `Point` is just a holder to the coordinates for each dimension in our space.

```
class Cluster:

  def __init__(self, center, points):
    self.center = center
    self.points = points
```

The `Cluster` is defined by its center and all other points it contains.

Given a path to image file we can create the points as follows:

[33] https://en.wikipedia.org/wiki/NP-hardness

```
def get_points(image_path):
  img = Image.open(image_path)
  img.thumbnail((200, 400))
  img = img.convert("RGB")
  w, h = img.size

  points = []
  for count, color in img.getcolors(w * h):
    for _ in range(count):
      points.append(Point(color))

  return points
```

A couple of things are happening here:

- load the image into memory
- resize it to smaller image (mobile UX requires big elements on the screen, so we aren't losing much color information)
- drop the alpha (transparency) information

Note that we're creating a `Point` for each pixel in our image.

Alright! You can now extract points from an image. But how can we calculate the distance between points in our clusters?

Distance function

Similar to the cost function in our supervised algorithm examples, we need a function that tells us how well we're doing. The objective of our algorithm is to minimize the distance between the points in each centroid.

One of the simplest distance functions we can use is the Euclidean distance, defined by:

$$d(p, q) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}$$

where p and q are two points from our space.

Note that while Euclidean distance is simple to implement it might not be the best way to calculate the color difference[34].

Here is the Python implementation:

[34] https://en.wikipedia.org/wiki/Color_difference

```
def euclidean(p, q):
  n_dim = len(p.coordinates)
  return sqrt(sum([
      (p.coordinates[i] - q.coordinates[i]) ** 2 for i in range(n_dim)
  ]))
```

Implementing K-Means clustering

Now that you have all pieces of the puzzle you can implement the K-Means clustering algorithm. Let's start with the method that finds the center for a set of points:

```
def calculate_center(self, points):
  n_dim = len(points[0].coordinates)
  vals = [0.0 for i in range(n_dim)]
  for p in points:
    for i in range(n_dim):
      vals[i] += p.coordinates[i]
  coords = [(v / len(points)) for v in vals]
  return Point(coords)
```

To find the center of a set of points, we add the values for each dimension and divide by the number of points.

Now for finding the actual clusters:

```
def assign_points(self, clusters, points):
  plists = [[] for i in range(self.n_clusters)]

  for p in points:
    smallest_distance = float('inf')

    for i in range(self.n_clusters):
      distance = euclidean(p, clusters[i].center)
      if distance < smallest_distance:
        smallest_distance = distance
        idx = i

    plists[idx].append(p)

  return plists

```

```
def fit(self, points):
  clusters = [Cluster(center=p, points=[p]) for p in random.sample(points, self.n_cl\
usters)]

  while True:

    plists = self.assign_points(clusters, points)

    diff = 0

    for i in range(self.n_clusters):
      if not plists[i]:
        continue
      old = clusters[i]
      center = self.calculate_center(plists[i])
      new = Cluster(center, plists[i])
      clusters[i] = new
      diff = max(diff, euclidean(old.center, new.center))

    if diff < self.min_diff:
      break

  return clusters
```

The implementation follows the description of the algorithm given above. Note that we exit the training loop when the color difference is lower than the one set by us.

Evaluation

Now that you have an implementation of K-Means clustering you can use it on the UI screenshots. We need a little more glue code to make it easier to extract color palettes:

```
def rgb_to_hex(rgb):
  return '#%s' % ''.join(('%02x' % p for p in rgb))

def get_colors(filename, n_colors=3):
  points = get_points(filename)
  clusters = KMeans(n_clusters=n_colors).fit(points)
  clusters.sort(key=lambda c: len(c.points), reverse = True)
  rgbs = [map(int, c.center.coordinates) for c in clusters]
  return list(map(rgb_to_hex, rgbs))
```

The `get_colors` function takes a path to an image file and the number of colors you want to extract from the image. We sort the clusters obtained from our algorithm based on the points in each (in descending order). Finally, we convert the RGB colors to hexadecimal values.

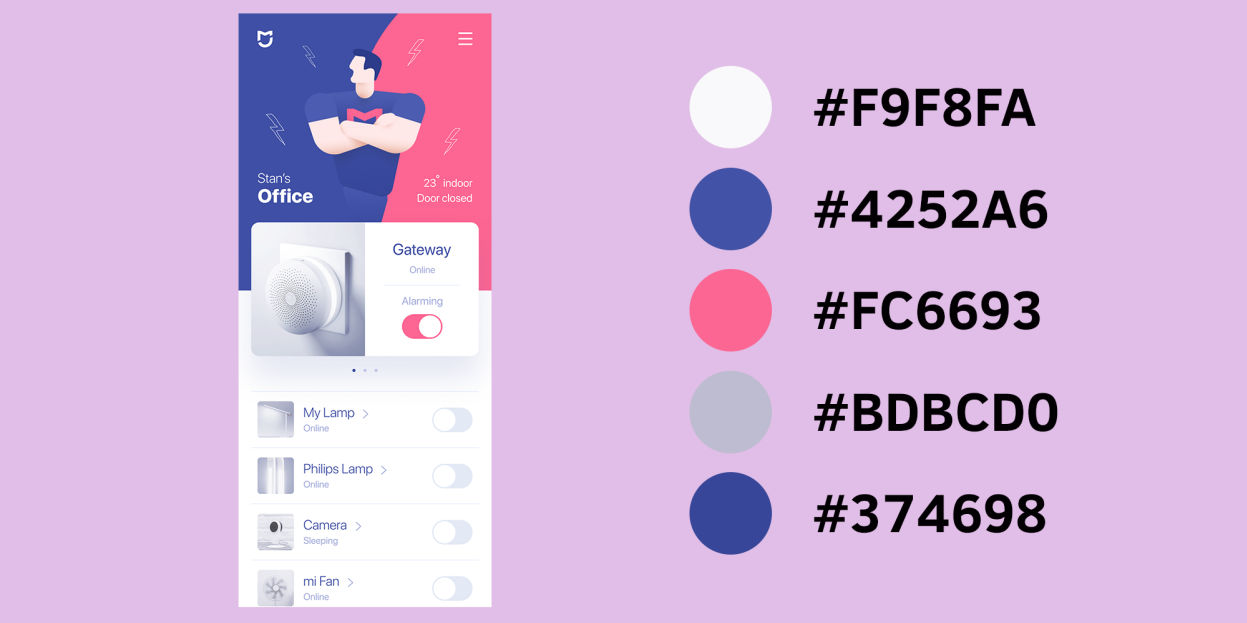
Let's extract the palette for the first UI screenshot from our data:

```
colors = get_colors('ui1.png', n_colors=5)
", ".join(colors)
```

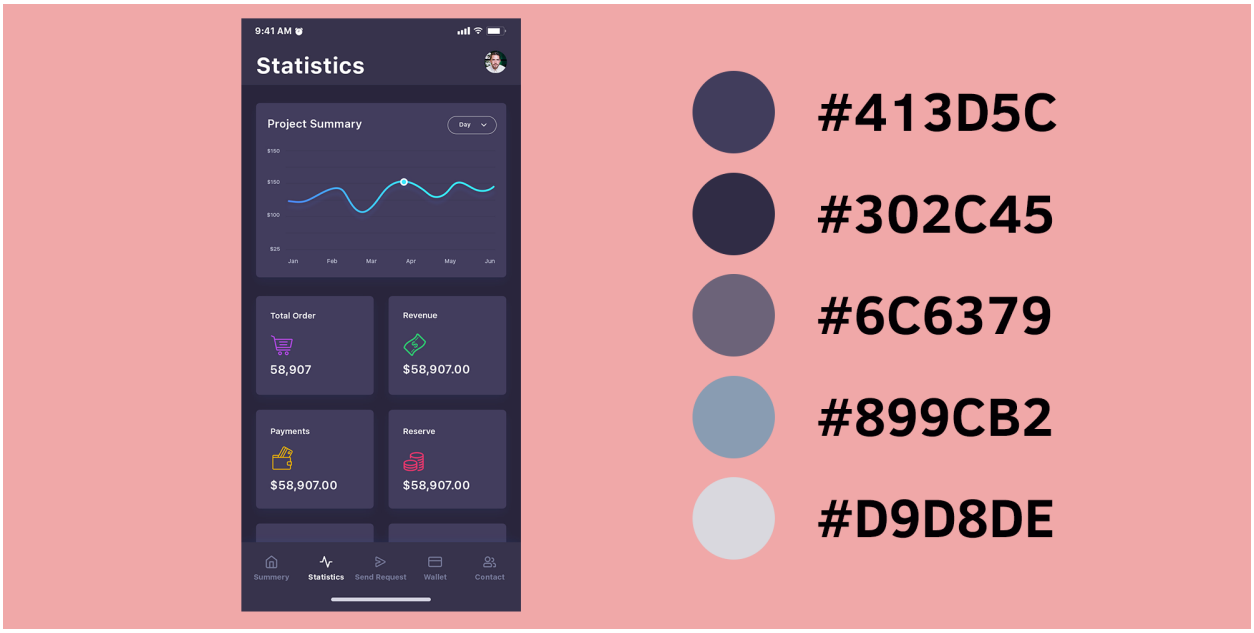
and here are the hexadecimal color values:

```
#f9f8fa, #4252a6, #fc6693, #bdbcd0, #374698
```

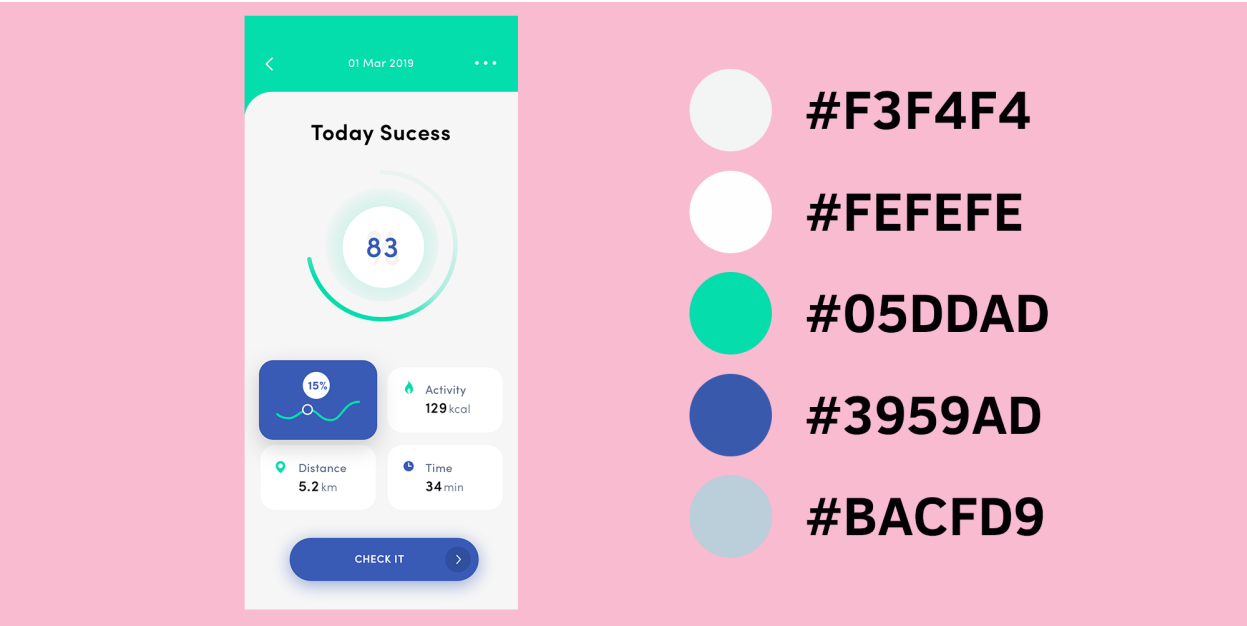
And for a visual representation of our results:



Running the clustering on the next two images, we obtain the following:



The last screenshot:



Well, that looks cool, right? Go on try it on your own images!

Complete source code notebook on Google Colaboratory[35]

[35] https://colab.research.google.com/drive/1_p1nptDfvJhcSvprmi1WtoIugsCI40Xa

Conclusion

Congratulations, you just implemented your first unsupervised algorithm from scratch! It also appears that it obtains good results when trying to extract a color palette from images.

Next up, we're going to do sentiment analysis on short phrases and learn a bit more how we can handle text data.

Movie review sentiment analysis with Naive Bayes

> TL;DR Build Naive Bayes text classification model using Python from Scratch. Use the model to classify IMDB movie reviews as positive or negative.

Textual data dominates our world from the tweets you read to the timeless writings of Seneca. And while we're consuming images (looking at you Instagram) and videos at increasing rates, you still read Google search results multiple times per daily.

One frequently recurring problem with text data is Sentiment analysis[36] (classification). Imagine you're a big sugar + water beverage company. You want to know what people think of your products. You'll search for texts with some tags, logos or names. You can then use Sentiment analysis to figure out if the opinions are positive or negative. Of course, you'll send the negative ones to your highly underpaid support center in India to sort things out.

Here, we'll build a generic text classifier that puts movie review texts into one of two categories - negative or positive sentiment. We're going to have a brief look at the Bayes theorem and relax its requirements using the Naive assumption.

Complete source code in Google Colaboratory Notebook[37]

Dealing with Text

Computers don't understand text data, though they do well with numbers. Natural Language Processing (NLP)[38] offers a set of approaches to solve text-related problems and represent text as numbers. While NLP is a vast field, we'll use some simple preprocessing techniques and Bag of Words[39] model.

The Data

Our data comes from a Kaggle challenge - "Bag of Words Meets Bags of Popcorn"[40].

We have 25,000 movie reviews from IMDB labeled as positive or negative. You might know that IMDB ratings are in the 0-10 range. An additional preprocessing step, done by the dataset authors, converts the rating to binary sentiment (<5 - negative). Of course, a single movie can have multiple reviews, but no more than 30.

[36] https://en.wikipedia.org/wiki/Sentiment_analysis
[37] https://colab.research.google.com/drive/1OPQDDJTKy0b40pziZWSsoBCmQV6HyXsm
[38] https://en.wikipedia.org/wiki/Natural_language_processing
[39] https://en.wikipedia.org/wiki/Bag-of-words_model
[40] https://www.kaggle.com/c/word2vec-nlp-tutorial

Reading the reviews

Let's load the training and test data in Pandas data frames:

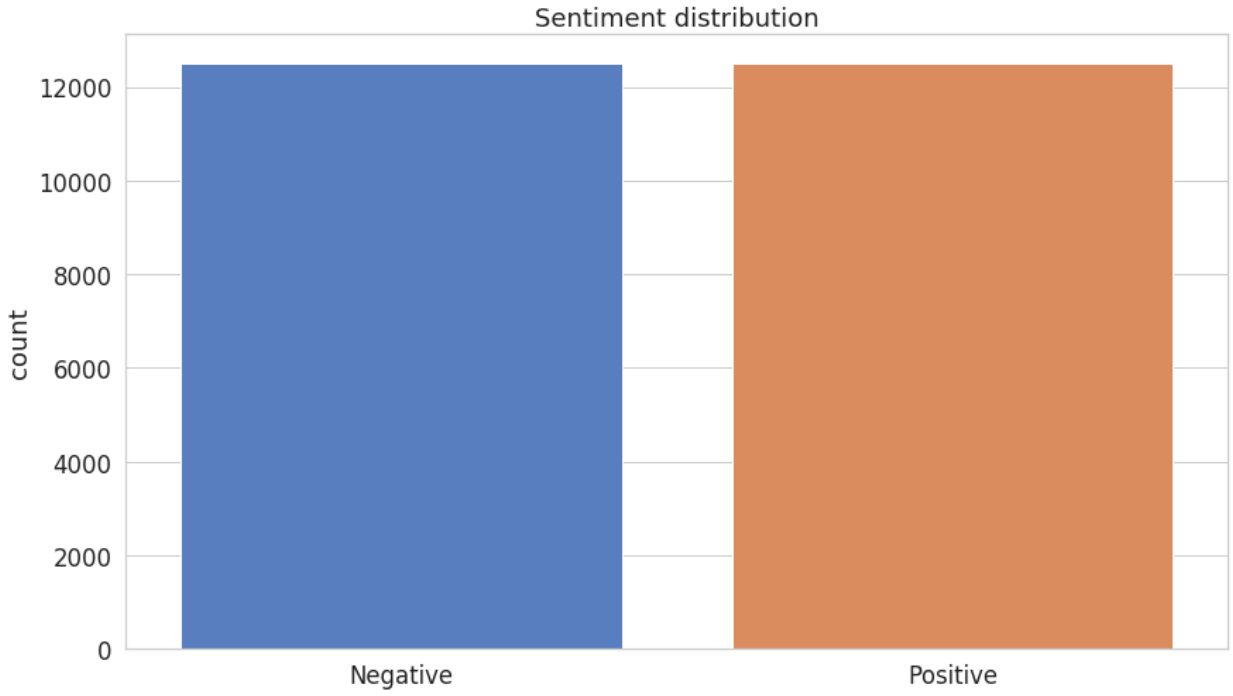
```
train = pd.read_csv("imdb_review_train.tsv", delimiter="\t")
test = pd.read_csv("imdb_review_test.tsv", delimiter="\t")
```

Exploration

Let's get a feel for our data. Here are the first 5 rows of the training data:

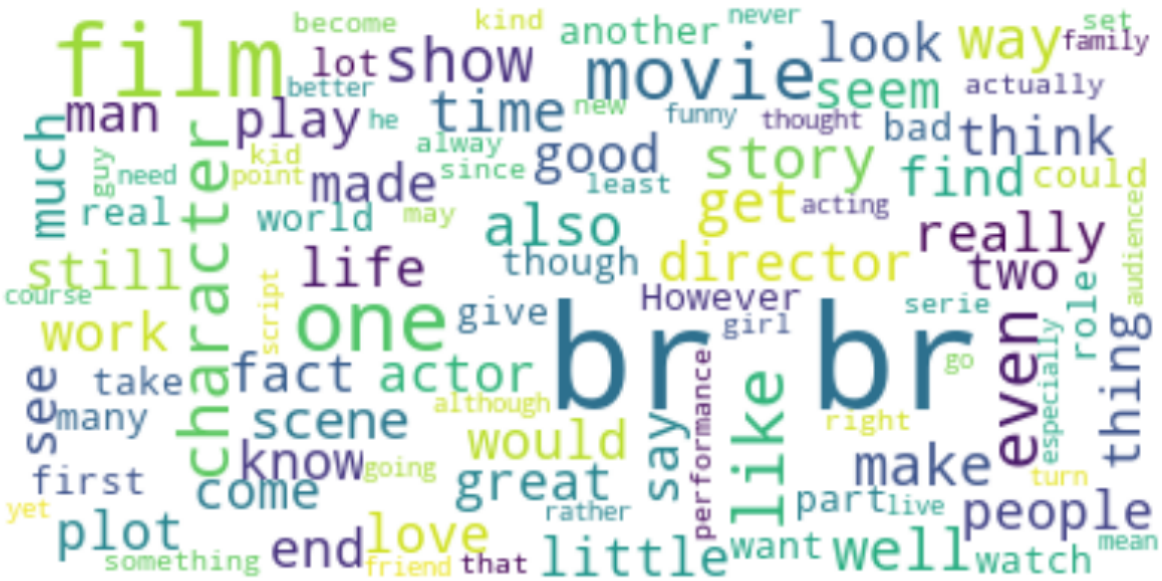
id	sentiment	review
5814_8	1	With all this stuff going down at the moment w...
2381_9	1	The Classic War of the Worlds" by Timothy Hi...
7759_3	0	The film starts with a manager (Nicholas Bell)...
3630_4	0	It must be assumed that those who praised this...
9495_8	1	Superbly trashy and wondrously unpretentious 8...

We're going to focus on the sentiment and review columns. The id column is combining the movie id with a unique number of a review. This might be a piece of important information in real-world scenarios, but we're going to keep it simple.



Both positive and negative sentiments have an equal presence. No need for additional gimmicks to fix that!

Here are the most common words in the training dataset reviews:



Hmm, that **br** looks weird, right?

Cleaning

Real-world text data is really messy. It can contain excessive punctuation, HTML tags (including that br), multiple spaces, etc. We'll try to remove/normalize most of it.

Most of the cleaning we'll do using Regular Expressions[41], but we'll use two libraries to handle HTML tags (surprisingly hard to remove) and removing common (stop) words:

```
def clean(self, text):
    no_html = BeautifulSoup(text).get_text()
    clean = re.sub("[^a-z\s]+", " ", no_html, flags=re.IGNORECASE)
    return re.sub("(\s+)", " ", clean)
```

First, we use BeautifulSoup[42] to remove HTML tags from our text. Second, we remove anything that is not a letter or space (note the ignoring of uppercase characters). Finally, we replace excessive spacing with a single one.

[41] https://en.wikipedia.org/wiki/Regular_expression
[42] https://www.crummy.com/software/BeautifulSoup/bs4/doc/

Tokenization

Now that our reviews are "clean", we can further prepare them for our Bag of Words model. Let's them to lowercase letters and split them into individual words. This process is known as tokenization[43]:

```
def tokenize(self, text):
    clean = self.clean(text).lower()
    stopwords_en = stopwords.words("english")
    return [w for w in re.split("\W+", clean) if not w in stopwords_en]
```

The last step of our pre-processing is to remove stop words[44] using those defined in the NLTK[45] library. Stop words are usually frequently occurring words that might not significantly affect the meaning of the text. Some examples in English are: "is", "the", "and".

An additional benefit of removing stop words is speeding up our models since we're removing the amount of train/test data. However, in real-world scenarios, you should think about whether removing stop words can be justified.

We'll place our clean and tokenize function in a class called *Tokenizer*.

Naive Bayes

Naive Bayes models are probabilistic classifiers that use the Bayes theorem[46] and make a strong assumption that the features of the data are independent. For our case, this means that each word is independent of others.

Intuitively, this might sound like a dumb idea. You know that (even from reading) the prev word(s) influence the current and next ones. However, the assumption simplifies the math and works really well in practice[47]!

The Bayes theorem is defined as:

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

where A and B are some events and $P(.)$ is probability.

This equation gives us the conditional probability of event A occurring given B has happened. In order to find this, we need to calculate the probability of B happening given A has happened and

[43] https://en.wikipedia.org/wiki/Lexical_analysis#Tokenization
[44] https://en.wikipedia.org/wiki/Stop_words
[45] https://www.nltk.org/
[46] https://en.wikipedia.org/wiki/Bayes%27_theorem
[47] https://www.cc.gatech.edu/~isbell/reading/papers/Rish.pdf

multiply that by the probability of A (known as *Prior*) happening. All of this is divided by the probability of B happening on its own.

The naive assumption allows us to reformulate the Bayes theorem for our example as:

$$P(Sentiment|w_1,\ldots,w_n) = \frac{P(Sentiment)\prod_{i=1}^{n} P(w_i|Sentiment)}{P(w_1,\ldots,w_n)}$$

We don't really care about probabilities. We only want to know whether a given text has a positive or negative sentiment. We can skip the denominator entirely since it just scales the numerator:

$$P(Sentiment|w_1,\ldots,w_n) \propto P(Sentiment)\prod_{i=1}^{n} P(w_i|Sentiment)$$

To choose the sentiment, we'll compare the scores for each sentiment and pick the one with a higher score.

Implementing Multinomial Naive Bayes

As you might've guessed by now, we're classifying text into one of two groups/categories - positive and negative sentiment.

Multinomial Naive Bayes allows us to represent the features of the model as frequencies of their occurrences (how often some word is present in our review). In other words, it tells us that the probability distributions we're using are multinomial[48].

Note on numerical stability

Our model relies on multiplying many probabilities. Those might become increasingly small and our computer might round them down to zero. To combat this, we're going to use log probability by taking log of each side in our equation:

$$\log P(Sentiment|w_1,\ldots,w_n) = \log P(Sentiment) + \log\prod_{i=1}^{n} P(w_i|Sentiment)$$

Note that we can still use the highest score of our model to predict the sentiment since log is monotonic[49].

[48] https://en.wikipedia.org/wiki/Multinomial_distribution
[49] https://en.wikipedia.org/wiki/Monotonic_function

Building our model

Finally, time to implement our model in Python. Let's start by defining some variables and group the data by class, so our training code is a bit tidier:

```
def fit(self, X, y):
    self.n_class_items = {}
    self.log_class_priors = {}
    self.word_counts = {}
    self.vocab = set()

    n = len(X)
    grouped_data = self.group_by_class(X, y)
    ...
```

We're going to implement a generic text classifier that doesn't assume the number of classes. You can use it for news category prediction, sentiment analysis, email spam detection, etc.

For each class, we'll find the number of examples in it and the log probability (prior). We'll also record the number of occurrences of each word and create a vocabulary - set of all words we've seen in the training data:

```
    for c, data in grouped_data.items():
      self.n_class_items[c] = len(data)
      self.log_class_priors[c] = math.log(self.n_class_items[c] / n)
      self.word_counts[c] = defaultdict(lambda: 0)

      for text in data:
        counts = Counter(self.tokenizer.tokenize(text))
        for word, count in counts.items():
            if word not in self.vocab:
                self.vocab.add(word)

            self.word_counts[c][word] += count
```

Note that we use our *Tokenizer* and the built-in class Counter[50] to convert a review to a bag of words.

In case you're interested, here's how `group_by_class` works:

[50] https://docs.python.org/3/library/collections.html#collections.Counter

```
def group_by_class(self, X, y):
  data = dict()
  for c in self.classes:
    data[c] = X[np.where(y == c)]
  return data
```

Making predictions

In order to predict sentiment from text data, we'll use our class priors and vocabulary:

```
def predict(self, X):
    result = []
    for text in X:

      class_scores = {c: self.log_class_priors[c] for c in self.classes}
      words = set(self.tokenizer.tokenize(text))

      for word in words:
          if word not in self.vocab: continue

          for c in self.classes:

            log_w_given_c = self.laplace_smoothing(word, c)
            class_scores[c] += log_w_given_c

      result.append(max(class_scores, key=class_scores.get))

    return result
```

For each review, we use the log priors for positive and negative sentiment and tokenize the text. For each word, we check if it is in the vocabulary and skip it if it is not. Then we calculate the log probability of this word for each class. We add the class scores and pick the class with a max score as our prediction.

Laplace smoothing

Note that $log(0)$ is *undefined* (and no, we're not using JavaScript here). It is entirely possible for a word in our vocabulary to be present in one class but not another - the probability of finding this word in that class will be 0! We can use Laplace smoothing[51] to fix this problem. We'll simply add 1 to the numerator but also add the size of our vocabulary to the denominator:

[51] https://en.wikipedia.org/wiki/Additive_smoothing

```
def laplace_smoothing(self, word, text_class):
  num = self.word_counts[text_class][word] + 1
  denom = self.n_class_items[text_class] + len(self.vocab)
  return math.log(num / denom)
```

Predicting sentiment

Now that your model can be trained and make predictions, let's use it to predict sentiment from movie reviews.

Data preparation

As discussed previously, we'll use only the review and sentiment columns from the training data. Let split it for training and testing:

```
X = train['review'].values
y = train['sentiment'].values

X_train, X_test, y_train, y_test = train_test_split(
  X, y,
  test_size=0.2,
  random_state=RANDOM_SEED
)
```

Evaluation

We'll pack our `fit` and `predict` functions into a class called `MultinomialNaiveBayes`. Let's use it:

```
MNB = MultinomialNaiveBayes(
    classes=np.unique(y),
    tokenizer=Tokenizer()
).fit(X_train, y_train)
```

Our classifier takes a list of possible classes and a Tokenizer as parameters. Also, the API is quite nice (thanks scikit-learn!)

```
y_hat = MNB.predict(X_test)
accuracy_score(y_test, y_hat)
```

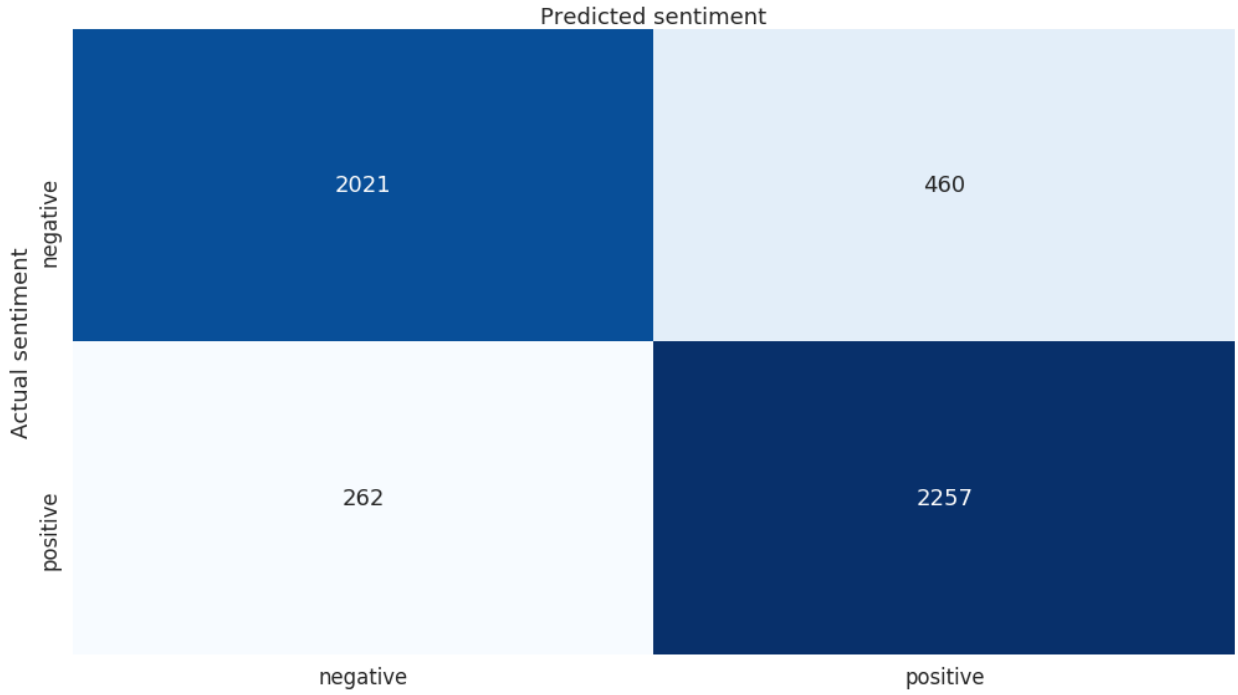
```
0.8556
```

This looks nice we got an accuracy of ~86% on the test set.

Here is the classification report:

```
              precision    recall  f1-score   support

           0       0.89      0.81      0.85      2481
           1       0.83      0.90      0.86      2519

    accuracy                           0.86      5000
   macro avg       0.86      0.86      0.86      5000
weighted avg       0.86      0.86      0.86      5000
```

And the confusion matrix:



Overall, our classifier does pretty well for himself. Submit the predictions to Kaggle and find out what place you'll get on the leaderboard.

Conclusion

Well done! You just built a Multinomial Naive Bayes classifier that does pretty well on sentiment prediction. You also learned about Bayes theorem, text processing, and Laplace smoothing! Will another flavor of the Naive Bayes classifier perform better?

Complete source code in Google Colaboratory Notebook[52]

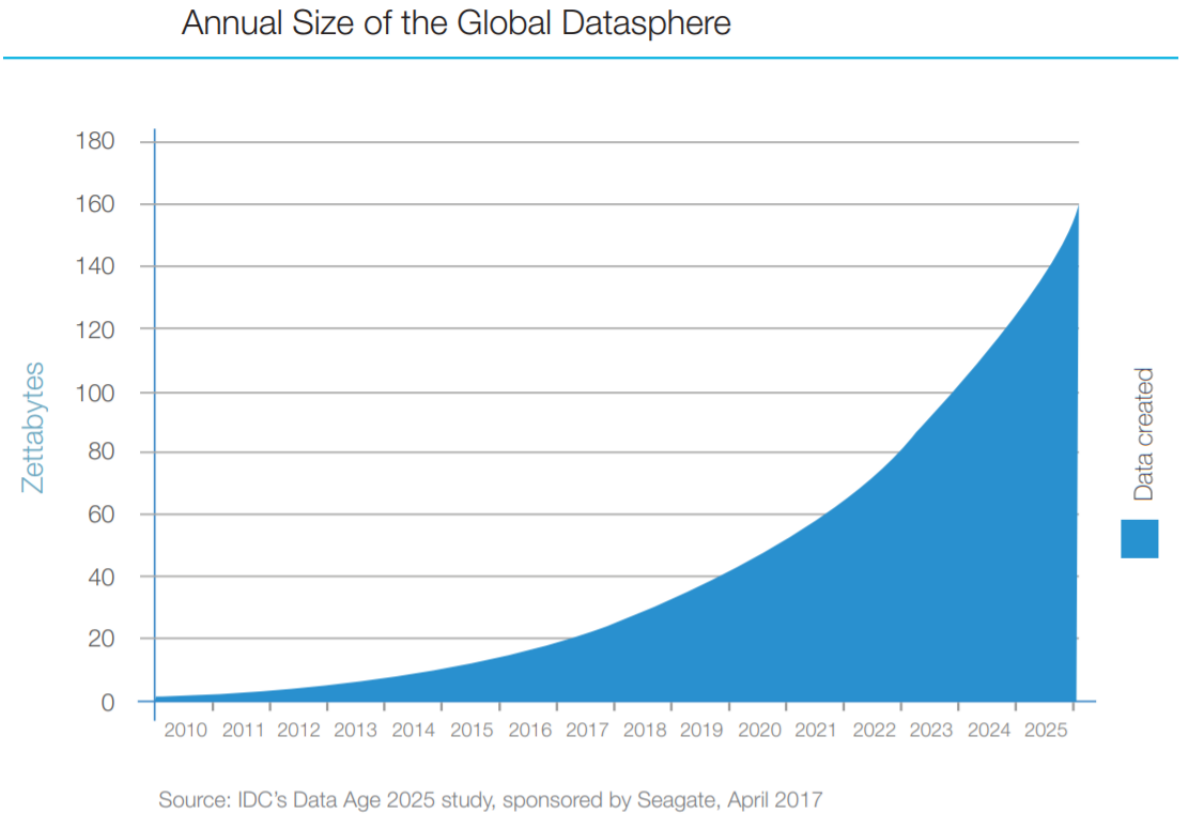
Next up, you'll build a recommender system from scratch!

[52] https://colab.research.google.com/drive/1OPQDDJTKy0b40pziZWSsoBCmQV6HyXsm

Music artist Recommender System using Stochastic Gradient Descent

> TL;DR Build a Recommender System in Python from scratch. Create a *ratings* matrix from last.fm dataset and preprocess the data. Train the model using Stochastic Gradient Descent (SGD) and use it to recommend music artists.

Recommender Systems are becoming more and more relevant as the amount of information on "The Internets" is exponentially increasing:



Finding what you might enjoy from books, movies, games to who to follow on Instagram becomes increasingly difficult. Moreover, we (the users) require faster interactions with the products we use on a daily basis, so we don't feel like we waste time (even though we do it more than ever before in

human history). As the amounts of data increase, your computational resources might have a hard time to produce fast enough results.

Here, we'll have a look at a succinct implementation of a Recommender System that is both the basis of many real-world implementations and is easy to understand. I can highly recommend you to read this through :)

Complete source code in Google Colaboratory Notebook[53]

User Ratings

Traditionally, recommender systems are built around user ratings given for a set of items, e.g. movie ratings on IMDB. Here, we'll have a look at using another metric for making recommendations.

Our data comes from last.fm[54], hosted by GroupLens (download from here[55]). It contains the following:

- **user_artists.dat** - userID, artistID, weight - plays of artist by user.
- **artists.dat** - id, name, url, pictureURL
- **tags.dat** - tagID, tagValue
- **user_taggedartists.dat** - userID, artistID, tagID, day, month, year
- **user_taggedartists-timestamps.dat** - userID, artistID, tagID, timestamp
- **user_friends.dat** - userID, friendID. User/friend relationships.

We'll focus on **user_artists.dat** and **artists.dat** as they contain all data required to make recommendations for new music artists to a user. Instead of ratings, we're going to use the play count by a user for each artist.

Loading the data

Let's load the data into Pandas data frames:

```
plays = pd.read_csv('user_artists.dat', sep='\t')
artists = pd.read_csv(
  'artists.dat',
  sep='\t',
  usecols=['id','name']
)
```

Data wrangling

You need to do a bit of wrangling before working with the data:

[53] https://colab.research.google.com/drive/1_WxDPLGkJY3qJ-PK0J1YjATaZz35efmk
[54] https://www.last.fm/
[55] https://grouplens.org/datasets/hetrec-2011/

```
ap = pd.merge(
  artists, plays,
  how="inner",
  left_on="id",
  right_on="artistID"
)
ap = ap.rename(columns={"weight": "playCount"})
```

We merge the artists and user plays and rename the weight column to *playCount*. Let's rank the artists based on how much they were played by the users:

```
artist_rank = ap.groupby(['name']) \
  .agg({'userID' : 'count', 'playCount' : 'sum'}) \
  .rename(columns={"userID" : 'totalUniqueUsers', "playCount" : "totalArtistPlays"})\
 \
  .sort_values(['totalArtistPlays'], ascending=False)

artist_rank['avgUserPlays'] = artist_rank['totalArtistPlays'] / artist_rank['totalUn\
iqueUsers']
```

And merge the results with the previous data frame:

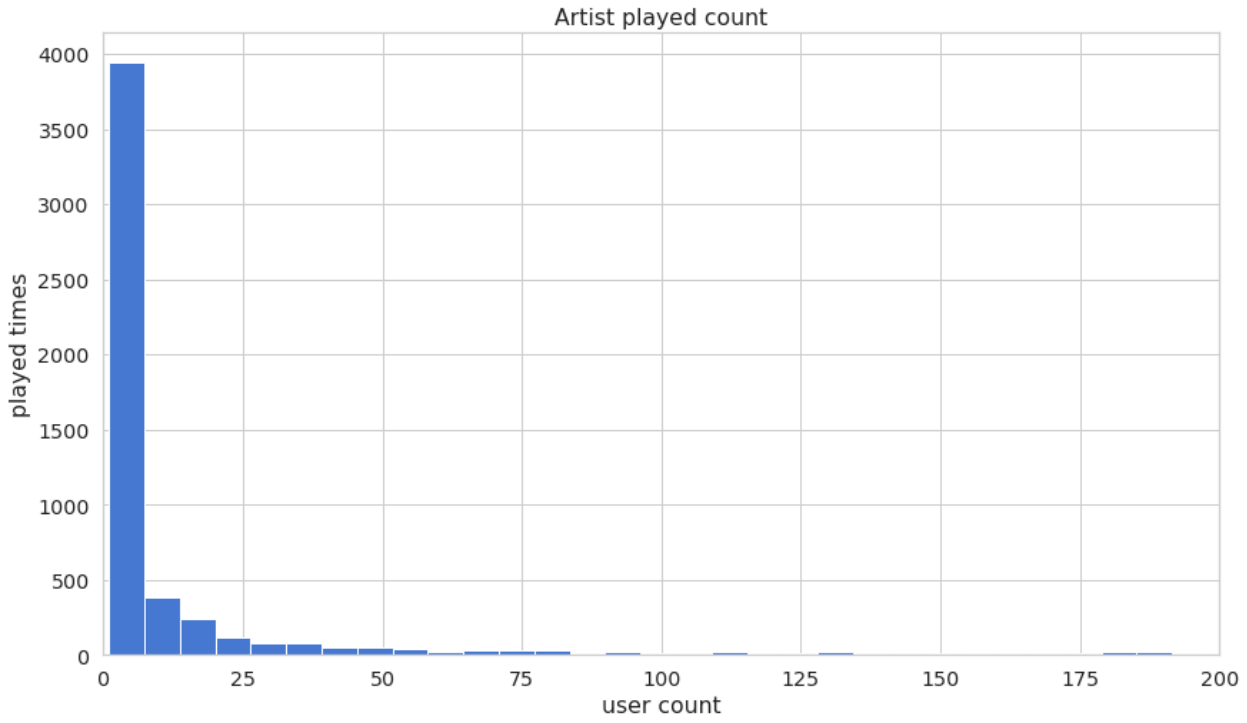
```
ap = ap.join(artist_rank, on="name", how="inner") \
  .sort_values(['playCount'], ascending=False)
```

Here is a subset of the data:

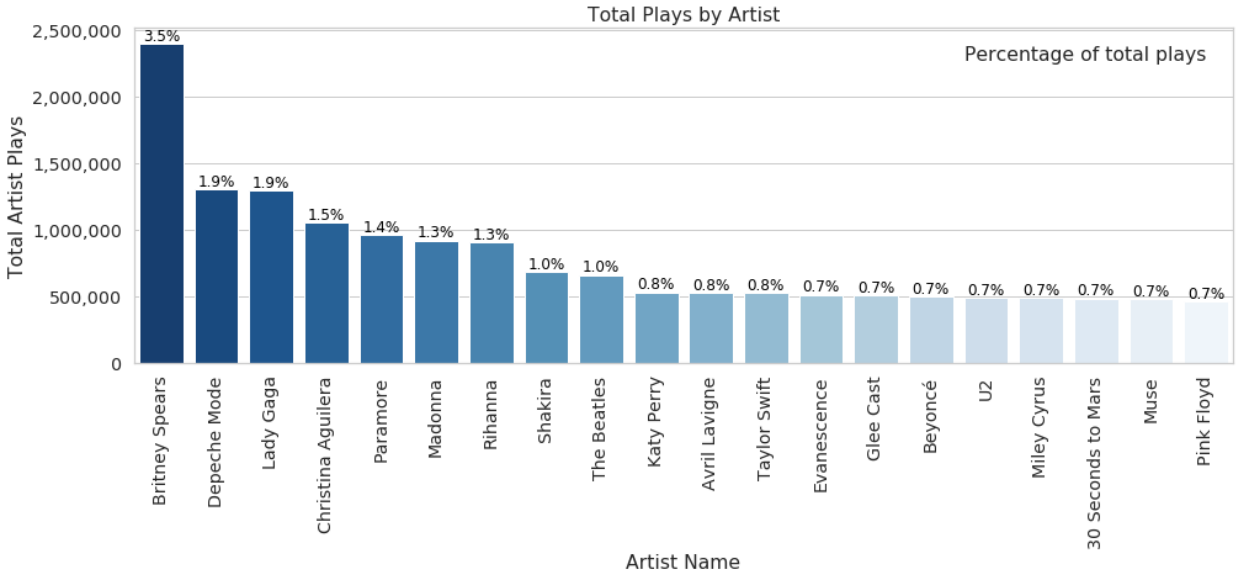
name	userID	artistID	playCount
Depeche Mode	1642	72	352698
Thalía	2071	792	324663
U2	1094	511	320725
Blur	1905	203	257978
Paramore	1664	498	227829

Exploration

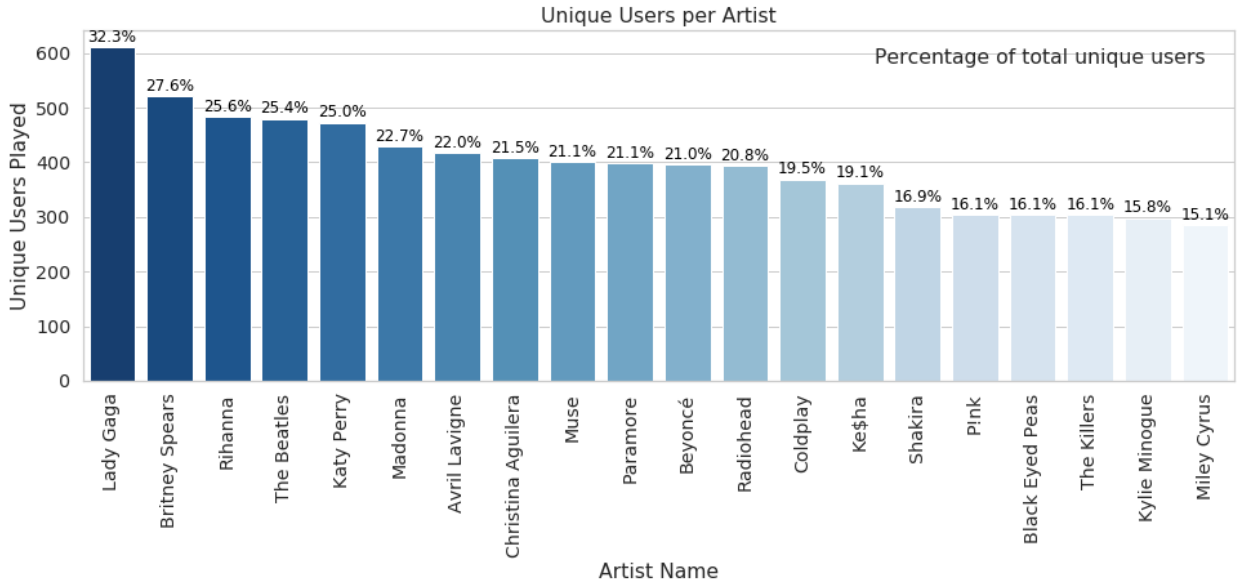
Let's look at how much each artist is played by users:



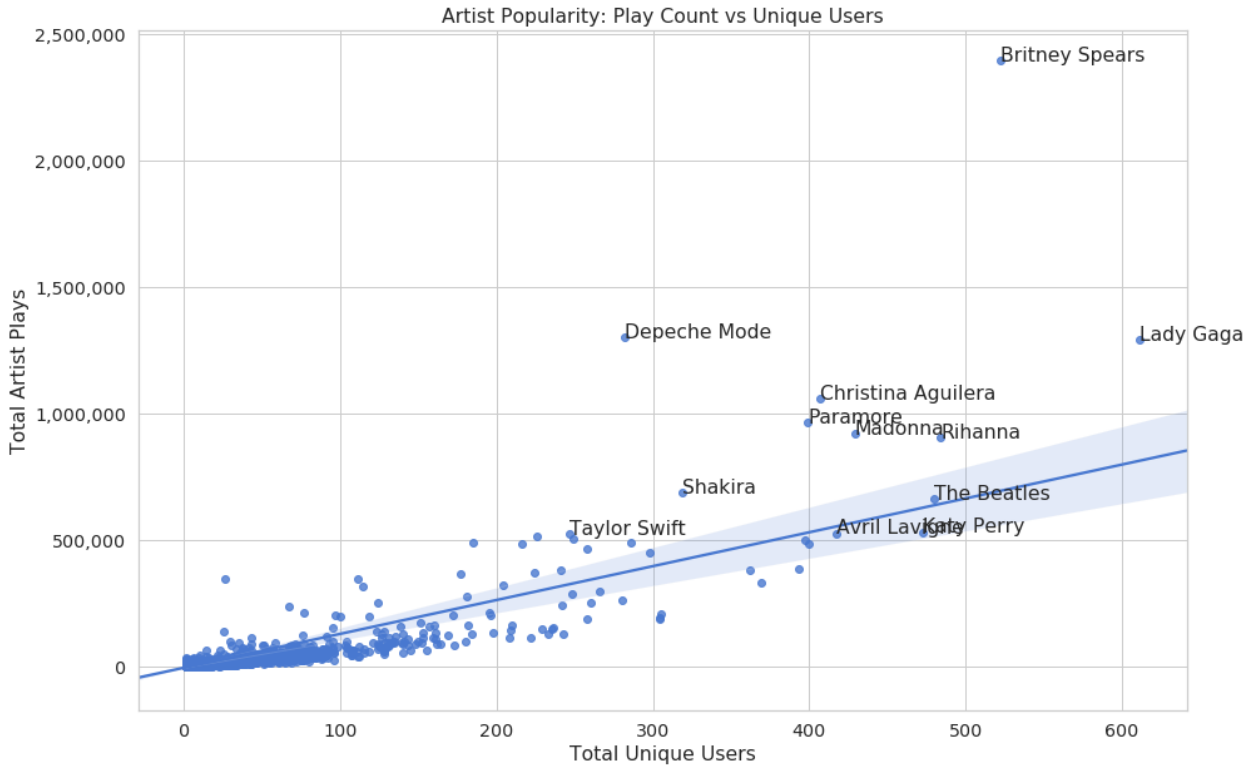
and the names of the artists that were played most:



how much of the users play the artist:



here's another look at artist popularity:



No surprises here, popular artists are taking the majority of the plays. Good thing that "The Beatles" still hold strong, though :)

Recommender Systems

Recommender systems (RS) are used pretty much everywhere where you have an array of items to choose from. They work by making suggestions that might help you make better/faster choices.

You might think that you're a special snowflake, but RS exploit behavior patterns that are shared between users (you and other people). For example, you and your friends might have similar tastes for some stuff. RS try to find "friends" within other users and recommend things you haven't tried that.

The two most used types of RS are Content-Based and Collaborative Filtering (CF). Collaborative filtering produces recommendations based on the knowledge of users' preference to items, that is it uses the "wisdom of the crowd" to recommend items. In contrast, content-based recommender systems focus on the properties of the items and give you recommendations based on the similarity between them.

Collaborative Filtering (CF)

Collaborative filtering (CF) is the workhorse of recommender engines. What is good about the algorithm is its property of being able to do feature learning, which allows it to start to learn which features to use. CF can be divided into *Memory-Based Collaborative Filtering* and *Model-Based Collaborative Filtering*.

Memory-Based Collaborative Filtering

Memory-Based CF methods can be divided into two sections: user-item filtering and item-item filtering. Here is the difference:

- Item-Item Collaborative Filtering: "Users who liked this item also liked …"
- User-Item Collaborative Filtering: "Users who are similar to you (kinda like the twin you never knew you had) also liked …"

Both methods require user-item matrix that contain the ratings for user u for item i. From that, you can calculate the similarity matrix.

The similarity values in Item-Item Collaborative Filtering are calculated by taking into account all users who have rated a pair of items.

For User-Item Collaborative Filtering, the similarity values are calculated by observing all items that are rated by a pair of users.

Model-Based Collaborative Filtering

Model-based CF methods are based on matrix factorization (MF)[56]. MF methods are used as an unsupervised learning method for latent variable decomposition and dimensionality reduction. They can handle scalability and sparsity problems better than Memory-based CF.

The goal of MF is to learn latent user preferences and item attributes from known ratings. Then use those variable to predict unknown ratings through the dot product of the latent features of users and items.

Matrix factorization restructures the user-item matrix into a low-rank matrix. You can represent it by the multiplication of two low-rank matrices, where the rows contain a vector of latent variables. You want this matrix to approximate the original matrix, as closely as possible, by multiplying the low-rank matrices together. That way, you predict the missing entries in the original matrix.

Singular Value Decomposition

Collaborative Filtering can be formulated by approximating a matrix X by using Singular Value Decomposition (SVD). The winning team at the Netflix Prize competition[57] used SVD matrix factorization models to win the prize. SVD can be expressed as:

$$X = USV^T$$

Given $m \times n$ matrix X:

- U is $(m \times r)$ orthogonal matrix
- S is $(r \times r)$ diagonal matrix with non-negative real numbers on the diagonal
- V^T is $(r \times n)$ orthogonal matrix

where U represents feature vectors of the users, V represents feature vectors of the items and the elements on the diagonal of S are known as *singular values*.

You can make a prediction by taking the dot product of U, S and V^T. Here is a quick example of how you can implement SVD in Python:

[56] https://en.wikipedia.org/wiki/Matrix_factorization_(recommender_systems)
[57] https://medium.com/netflix-techblog/netflix-recommendations-beyond-the-5-stars-part-1-55838468f429

```
import scipy.sparse as sp
from scipy.sparse.linalg import svds

U, S, VT = svds(user_item_ratings, k = 20)
S_diagonal = np.diag(S)
Y_hat = np.dot(np.dot(U, S_diagonal), VT)
```

Recommending music artists

While most tutorials on "the Internets" focus on memory-based approaches, they don't seem to be used in practice. Although they produce good results, they do not scale well and suffer from the "cold-start"[58] problem.

On the other hand, applying **SVD** requires factorization of the user-item matrix which can be expensive when the matrix is very sparse (lots of user-item ratings are missing). Also, imputing missing values is often used, since **SVD** doesn't work when data is missing. This can significantly increase the amount of data and runtime performance of the algorithm.

More recent approaches have focused on predicting ratings by minimizing the regularized squared error with respect to a latent user feature matrix P and a latent item feature matrix Q:

$$\min_{Q*,P*} \sum_{(u,i)\in K} (r_{ui} - P_u^T Q_i)^2 + \lambda(||Q_i||^2 + ||P_u||^2)$$

Where K is a set of (u, i) pairs, $r(u, i)$ is the rating for item i by user u and λ is a regularization term (used to avoid overfitting). The training of our model consists of minimizing the regularized squared error. After an estimate of P and Q is obtained, you can predict unknown ratings by taking the dot product of the latent features for users and items.

We can apply Stochastic Gradient Descent (SGD)[59] or Alternating Least Squares (ALS)[60] in order to minimize the loss function. Both methods can be used to incrementally update our model (online learning), as new rating comes in.

Here, we'll implement **SGD** as it seems to be generally faster and more accurate[61] than ALS (except in situations of highly sparse and implicit data). As an added bonus, **SGD** is widely used for training Deep Neural Networks (which we'll discuss later). Thus, many high-quality implementations exist for this algorithm.

[58] https://en.wikipedia.org/wiki/Cold_start_(computing)
[59] https://en.wikipedia.org/wiki/Stochastic_gradient_descent
[60] https://www.quora.com/What-is-the-Alternating-Least-Squares-method-in-recommendation-systems-And-why-does-this-algorithm-work-intuition-behind-this
[61] http://cs229.stanford.edu/proj2014/Christopher%20Aberger,%20Recommender.pdf

Preprocessing

In order to apply the CF algorithm, we need to transform our dataset into a user-artist play count matrix. Let's do that after doing data scaling first:

```
pc = ap.playCount
play_count_scaled = (pc - pc.min()) / (pc.max() - pc.min())

ap = ap.assign(playCountScaled=play_count_scaled)
```

This squishes the play counts in the [0 - 1] range and adds a new column to our data frame. Let's build our "ratings" data frame:

```
ratings_df = ap.pivot(
    index='userID',
    columns='artistID',
    values='playCountScaled'
)

ratings = ratings_df.fillna(0).values
```

We use Pandas pivot[62] method to create index/column data frame and fill the missing play counts with *0*.

Let's have a look at how sparse our data frame is:

```
sparsity = float(len(ratings.nonzero()[0]))
sparsity /= (ratings.shape[0] * ratings.shape[1])
sparsity *= 100
print('{:.2f}%'.format(sparsity))
```

```
0.28%
```

Our dataset seems really sparse. Next, let's split our data into training and validation data sets:

```
train, val = train_test_split(ratings)
```

Here, we define the `train_test_split` function a bit differently:

[62] https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.pivot.html

```
MIN_USER_RATINGS = 35
DELETE_RATING_COUNT = 15

def train_test_split(ratings):

    validation = np.zeros(ratings.shape)
    train = ratings.copy()

    for user in np.arange(ratings.shape[0]):
        if len(ratings[user,:].nonzero()[0]) >= MIN_USER_RATINGS:
            val_ratings = np.random.choice(
              ratings[user, :].nonzero()[0],
              size=DELETE_RATING_COUNT,
              replace=False
            )
            train[user, val_ratings] = 0
            validation[user, val_ratings] = ratings[user, val_ratings]
    return train, validation
```

Okay, that is a lot different than you might've expected. We remove some existing play counts from users by replacing them with zeros.

Measuring the error

One of the most popular metrics used to evaluate the accuracy of Recommender Systems is Root Mean Squared Error (RMSE), defined as:

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum(y_i - \hat{y}_i)^2}$$

where y_i is the real value for item i, $\hat{y}_i$ is the predicted one and N is the size of the training set.

A discussion on more evaluation metrics can be found here[63]

Here is the *RMSE* in Python:

```
def rmse(prediction, ground_truth):
    prediction = prediction[ground_truth.nonzero()].flatten()
    ground_truth = ground_truth[ground_truth.nonzero()].flatten()
    return sqrt(mean_squared_error(prediction, ground_truth))
```

Training

Let's use SGD to train our recommender:

[63] https://www.microsoft.com/en-us/research/publication/evaluating-recommender-systems/

```
def fit(self, X_train, X_val):
  m, n = X_train.shape

  self.P = 3 * np.random.rand(self.n_latent_features, m)
  self.Q = 3 * np.random.rand(self.n_latent_features, n)

  self.train_error = []
  self.val_error = []

  users, items = X_train.nonzero()

  for epoch in range(self.n_epochs):
      for u, i in zip(users, items):
          error = X_train[u, i] - self.predictions(self.P[:,u], self.Q[:,i])
          self.P[:, u] += self.learning_rate * \
           (error * self.Q[:, i] - self.lmbda * self.P[:, u])
          self.Q[:, i] += self.learning_rate * \
           (error * self.P[:, u] - self.lmbda * self.Q[:, i])

      train_rmse = rmse(self.predictions(self.P, self.Q), X_train)
      val_rmse = rmse(self.predictions(self.P, self.Q), X_val)
      self.train_error.append(train_rmse)
      self.val_error.append(val_rmse)
```

We start by creating 2 matrices for the latent features of the users and ratings. For each user, item pair, we calculate the error (note we use a simple difference of the existing and predicted ratings). We then update P and Q using Gradient Descent.

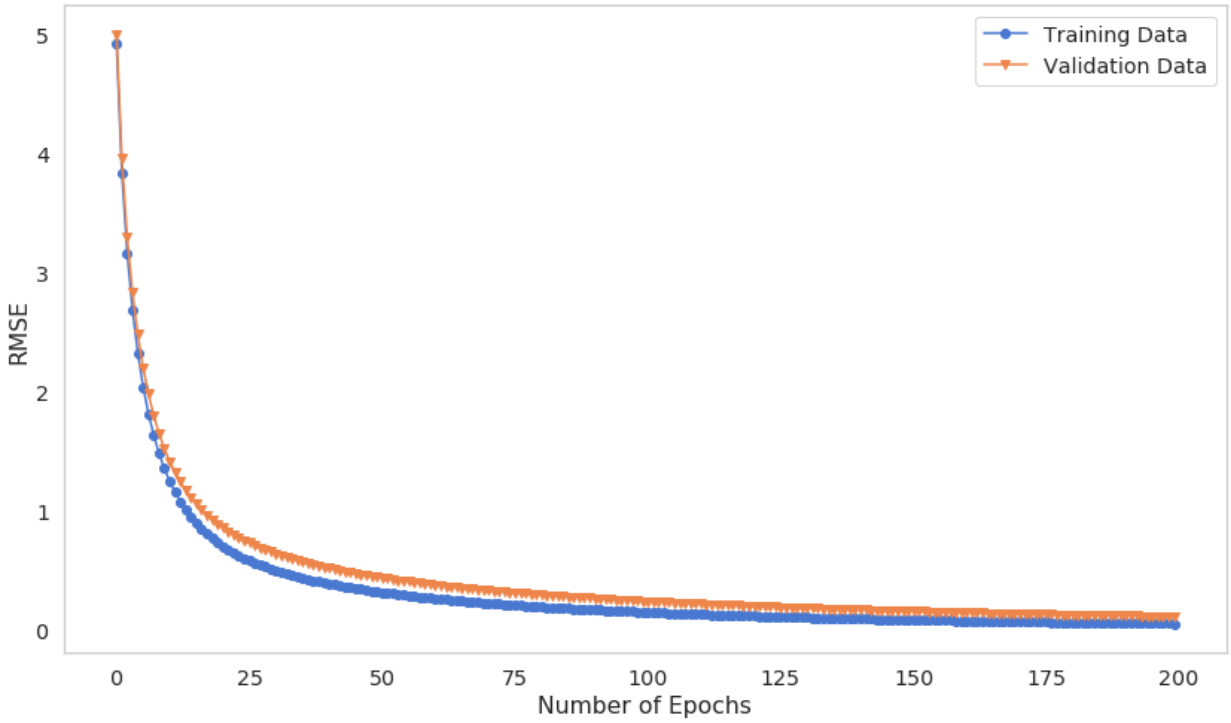
After each training epoch, we calculate the training and validation errors and store their values to analyze later. Here is the `predictions` method implementation:

```
def predictions(self, P, Q):
  return np.dot(P.T, Q)
```

As we discussed earlier, we obtain predictions by taking the dot product of the transposed P matrix with Q.

Evaluation

Let's have a quick look at how the training process went by looking at the training and validation RMSE:



Our model seems to train gradually, and both errors decrease as the number of epochs increase. Note that you might want to spend even more time to train your model, as it might get even better.

Making recommendations

Finally, we are ready to make some recommendations for a user. Here is the implementation of the `predict` method:

```
def predict(self, X_train, user_index):
  y_hat = self.predictions(self.P, self.Q)
  predictions_index = np.where(X_train[user_index, :] == 0)[0]
  return y_hat[user_index, predictions_index].flatten()
```

First, we obtain the matrix with all predicted play counts. Second, we take the indices of all unknown play counts and return only these as predictions. Let's make some recommendations for a specific user:

```
user_id = 1236
user_index = ratings_df.index.get_loc(user_id)
predictions_index = np.where(train[user_index, :] == 0)[0]

rating_predictions = recommender.predict(train, user_index)
```

Before looking at the recommendation list, let's have a look at what this user preferences currently are:

```
existing_ratings_index = np.where(train[user_index, :] > 0)[0]
existing_ratings = train[user_index, existing_ratings_index]

create_artist_ratings(
  artists,
  existing_ratings_index,
  existing_ratings
)
```

name	rating
Marilyn Manson	0.196486
3 Doors Down	0.043204
Pearl Jam	0.042016
Children of Bodom	0.025657
Disturbed	0.021690
Rammstein	0.021562
A Perfect Circle	0.020879
Gojira	0.017051
Rob Zombie	0.016280
D12	0.010990

And here are the recommendations from our model:

```
create_artist_ratings(
  artists,
  predictions_index,
  rating_predictions
)
```

name	rating
Sander van Doorn	0.559202
Jacob Miller	0.552893
Celtas Cortos	0.552142
Camisa de Vênus	0.546361
Ella Fitzgerald & Louis Armstrong	0.541477
The Answer	0.537367
Anjelika Akbar	0.536756
Medications	0.536486
Lützenkirchen	0.535515
Archie Star	0.535147

Now might be a good time to go on YouTube or Spotify and try out some of those artists!

Conclusion

Congratulations on building a highly performant Recommender System from scratch! You've learned how to:

- Prepare raw data for making recommendations
- Implemented a simple Stochastic Gradient Descent from scratch
- Used the model to make predictions for new artists you might actually enjoy!

Complete source code in Google Colaboratory Notebook[64]

Can you apply this same model on your dataset? Tell me how it went in the comments below!

[64] https://colab.research.google.com/drive/1_WxDPLGkJY3qJ-PK0J1YjATaZz35efmk

Fashion product image classification using Neural Networks

> TL;DR Build Neural Network in Python from scratch. Use the model to classify images of fashion products into 1 of 10 classes.

We live in the age of Instagram, YouTube, and Twitter. Images and video (a sequence of images) dominate the way millennials and other weirdos consume information.

Having models that understand what images show can be crucial for understanding your emotional state (yes, you might get a personalized Coke ad right after you post your breakup selfie on Instagram), location, interests and social group.

Predominantly, models that understand image data used in practice are (Deep) Neural Networks. Here, we'll implement a Neural Network image classifier from scratch in Python.

Complete source code in Google Colaboratory Notebook[65]

Image Data

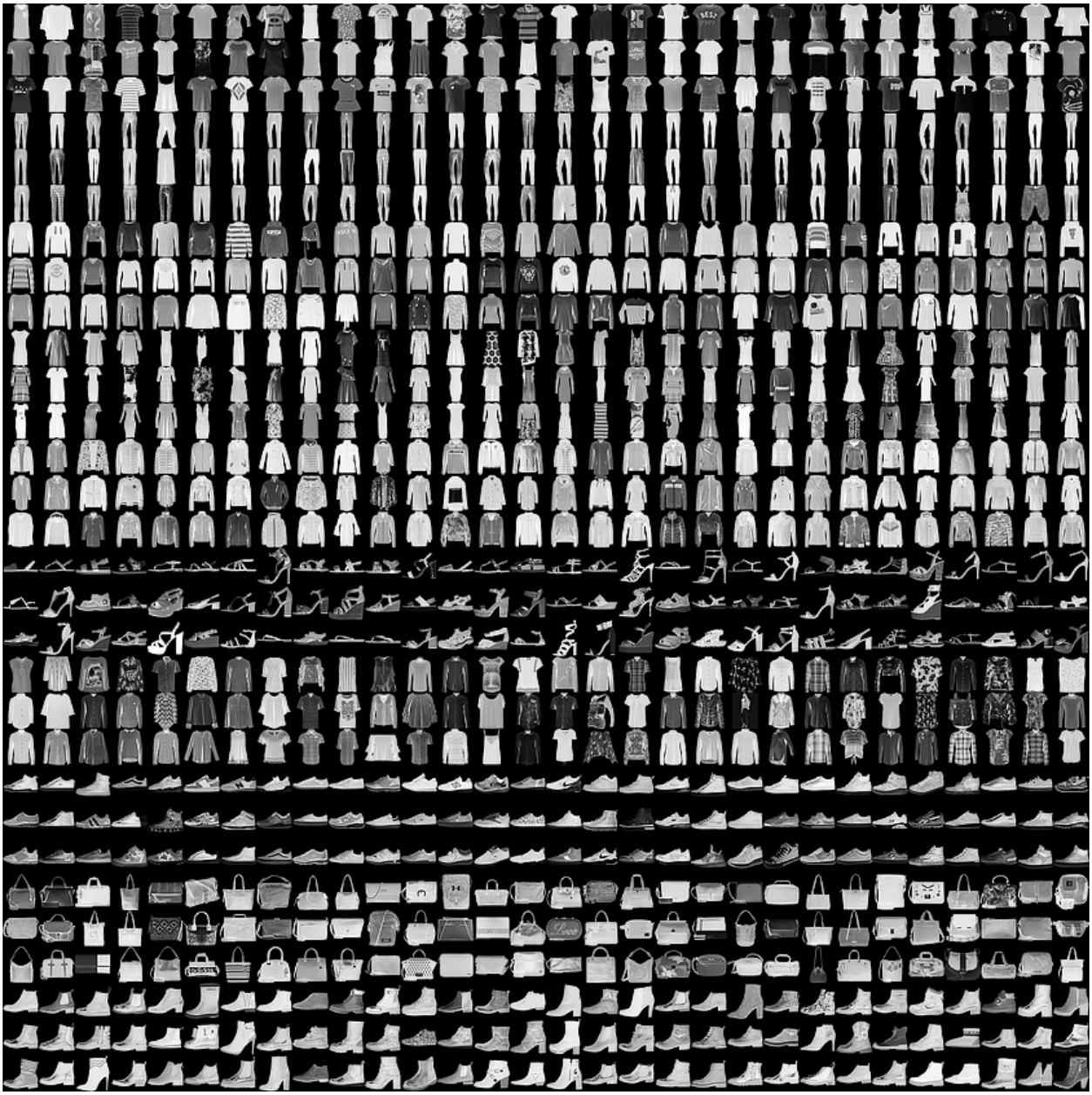
Hopefully, it's not a complete surprise to you that computers can't actually see images as we do. Each image on your device is represented/stored as a matrix, where each pixel is one or more numbers.

Reading the fashion products data

> Fashion-MNIST[66] is a dataset of Zalando's article images—consisting of a training set of *60,000* examples and a test set of *10,000* examples. Each example is a *28x28* grayscale image, associated with a label from *10* classes. We intend Fashion-MNIST to serve as a direct drop-in replacement for the original MNIST dataset for benchmarking machine learning algorithms. It shares the same image size and structure of training and testing splits.

Here is a sample of the images:

[65] https://drive.google.com/file/d/1S59KWV8KmZTI-A6OpXge3yG87HXyIcUz/view?usp=sharing
[66] https://github.com/zalandoresearch/fashion-mnist

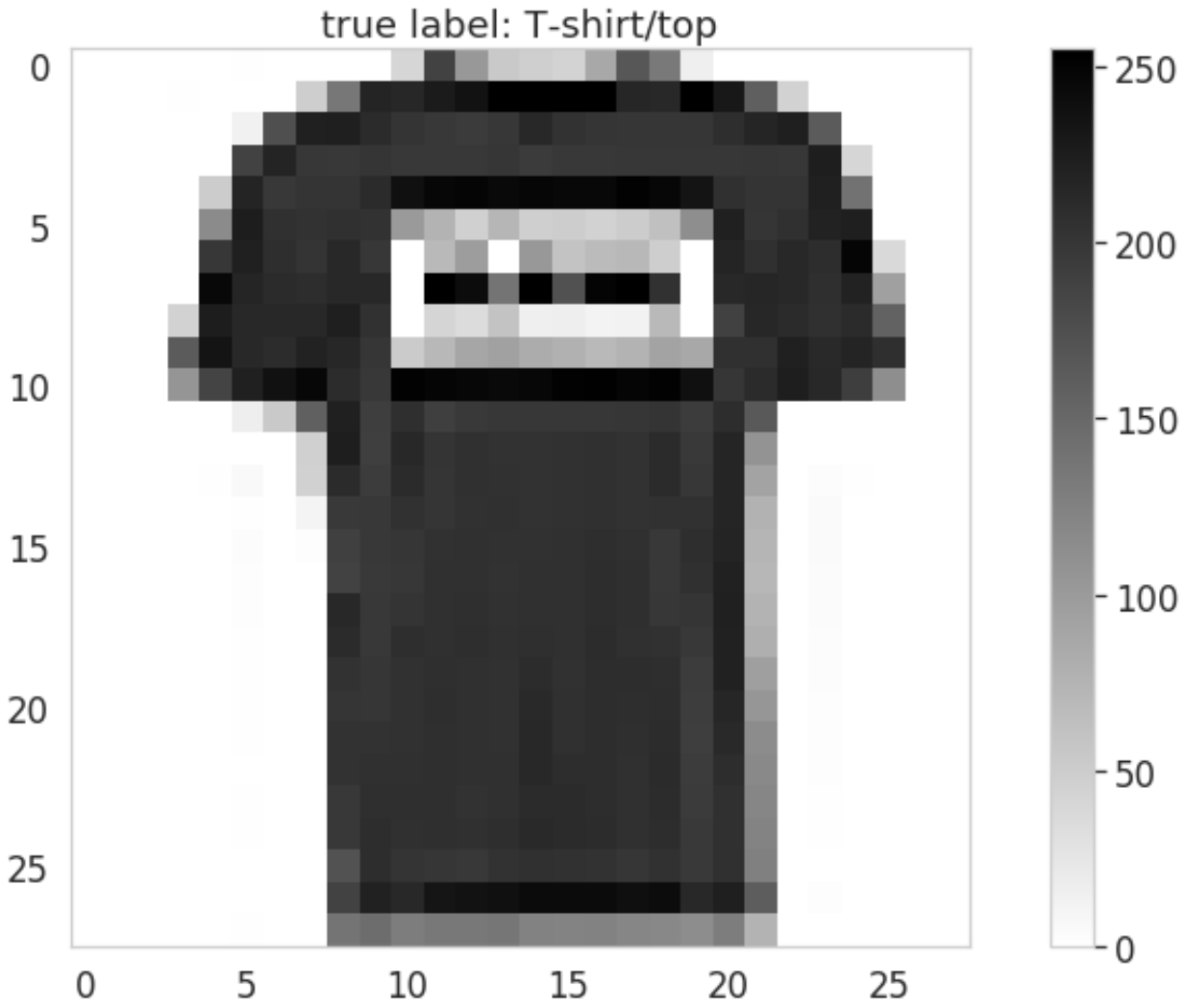


You might be familiar with the original handwritten digits MNIST[67] dataset and wondering why we're not using it? Well, it might be too easy to make predictions on[68]. And of course, fashion is cooler, right?

Exploration

The product images are grayscale, 28x28 pixels and look something like this:

[67] http://yann.lecun.com/exdb/mnist/
[68] http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/



Here are the first 3 rows from the pixel matrix of the image:

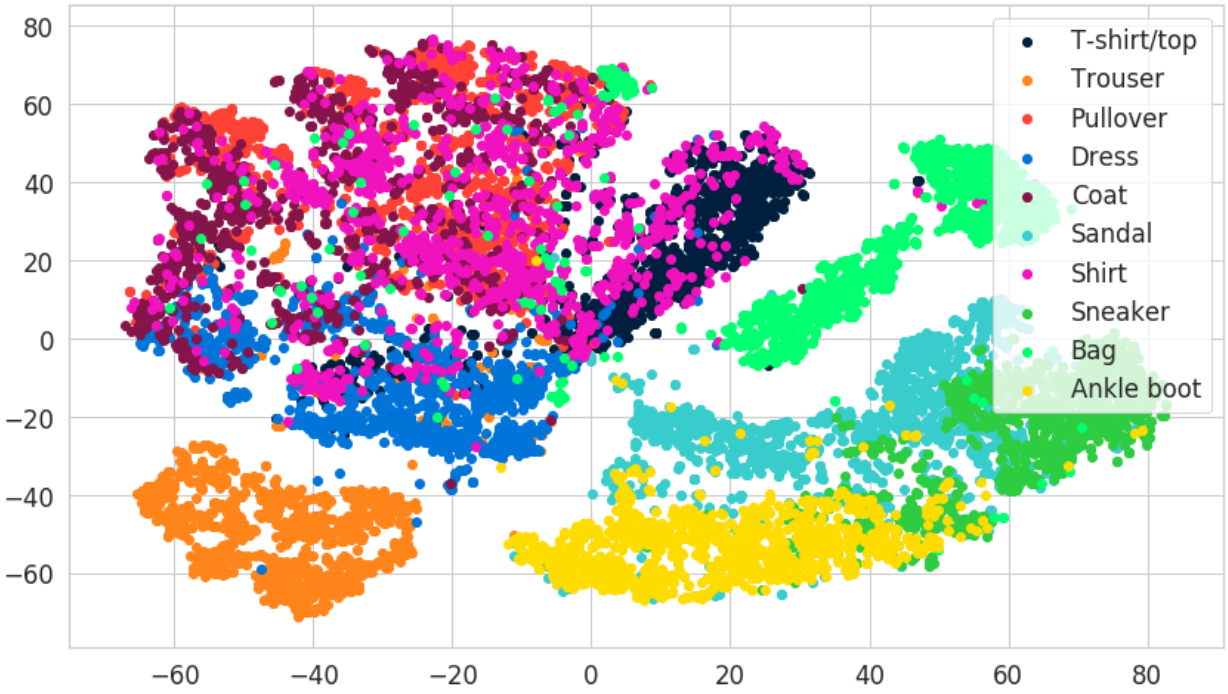
```
[[  0,   0,   0,   0,   0,   1,   0,   0,   0,   0,  41, 188, 103,
     54,  48,  43,  87, 168, 133,  16,   0,   0,   0,   0,   0,   0,
      0,   0],
   [  0,   0,   0,   1,   0,   0,   0,  49, 136, 219, 216, 228, 236,
    255, 255, 255, 255, 217, 215, 254, 231, 160,  45,   0,   0,   0,
      0,   0],
   [  0,   0,   0,   0,   0,  14, 176, 222, 224, 212, 203, 198, 196,
    200, 215, 204, 202, 201, 201, 201, 209, 218, 224, 164,   0,   0,
      0,   0],
   ...
]
```

Note that the values are in the 0-255 range (grayscale).

We have 10 classes of possible fashion products:

```
['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
 'Sandal',      'Shirt',   'Sneaker',  'Bag',   'Ankle boot']
```

Let's have a look at a lower dimensional representation of some of the products using t-SNE[69]. We'll transform the data into 2-dimensional using the implementation from scikit-learn[70]:



You can observe a clear separation between some classes and significant overlap between others. Let's build a Neural Network that can try to separate between different fashion products!

Neural Networks

Neural Networks (NNs), Deep Neural Networks in particular, are all the rage in the last couple of years in the Machine Learning realm. That's hardly a surprise since most state-of-the-art results (SOTA)[71] on various Machine Learning problems are obtained via Neural Nets.

> Browse papers with source code achieving SOTA[72]

[69] https://en.wikipedia.org/wiki/T-distributed_stochastic_neighbor_embedding
[70] https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html
[71] https://github.com/RedditSota/state-of-the-art-result-for-machine-learning-problems
[72] https://paperswithcode.com/sota

The Artificial Neuron

The goal of modeling our biological neuron has led to the invention of the artificial neuron. Here is how a single neuron in your brain looks like:

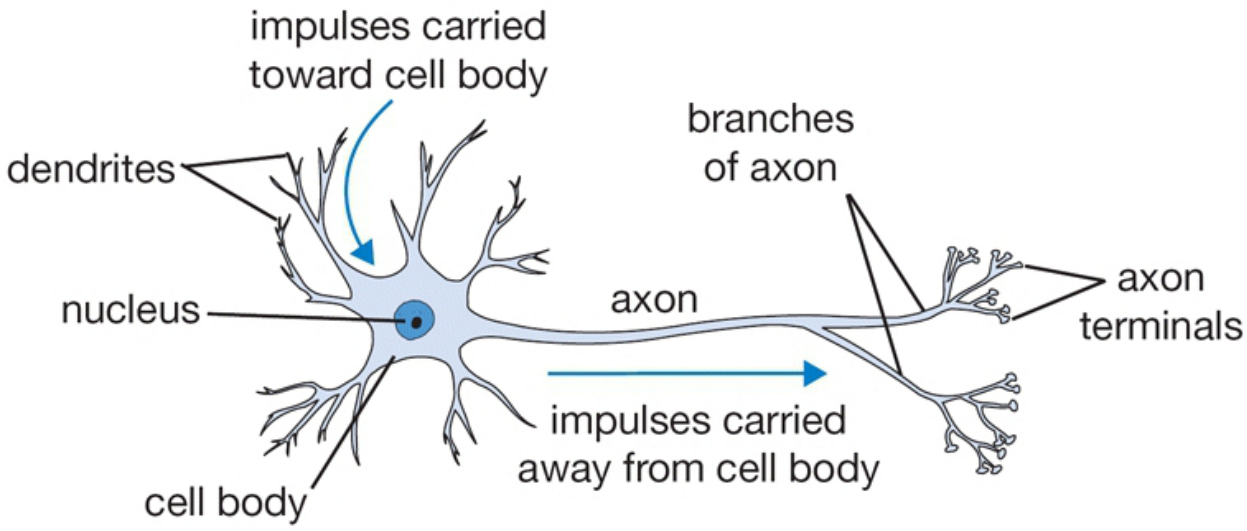


Image credit: CS231n[73]

On the other side, we have a vastly simplified mathematical model that turns out to be extremely useful in practice (as evident by the success of Neural Nets):

[73] https://cs231n.github.io/

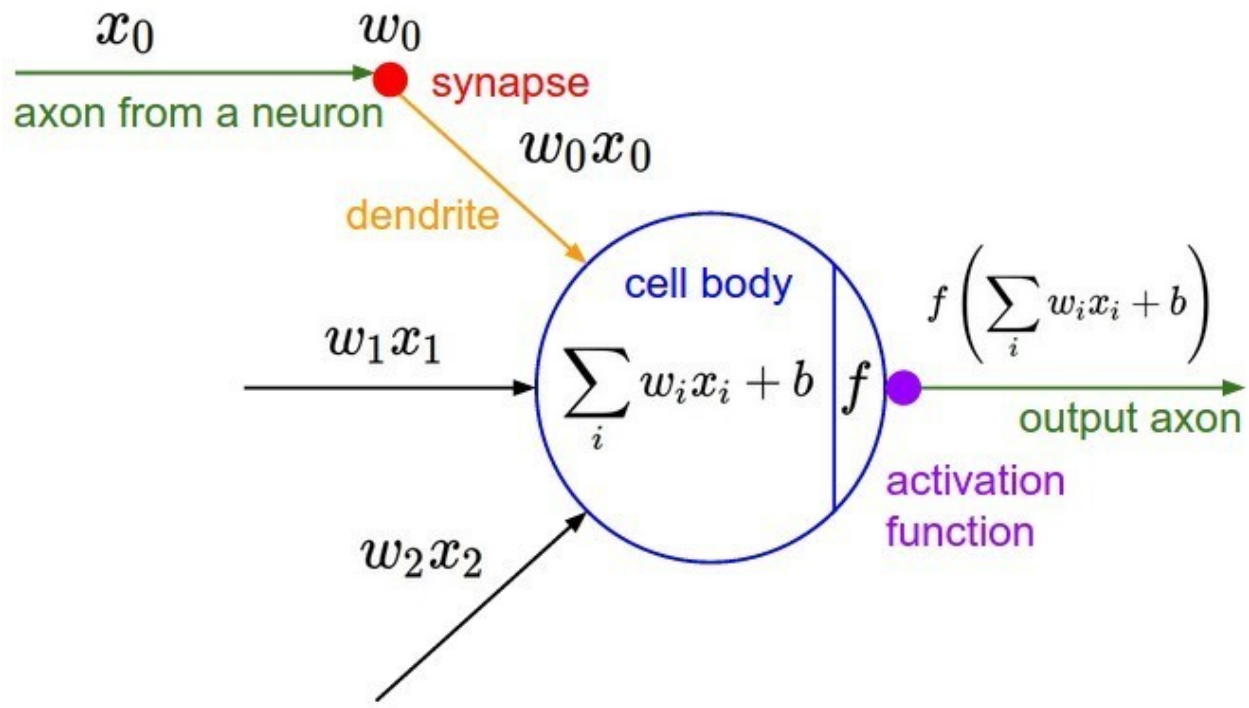


Image credit: CS231n[74]

The idea of the artificial neuron is simple - you have data vector X coming from somewhere, a vector of parameters W and a bias vector b. The output of a neuron is given by:

$$Y = f\left(\sum_i w_i x_i + b\right)$$

where f is an activation function that controls how strong the output signal of the neuron is.

Architecting Neural Networks

You can use a single neuron as a classifier, but the fun part begins when you group them into layers. Concretely, the neurons are connected into an acyclic graph with the data flowing between layers:

[74] https://cs231n.github.io/

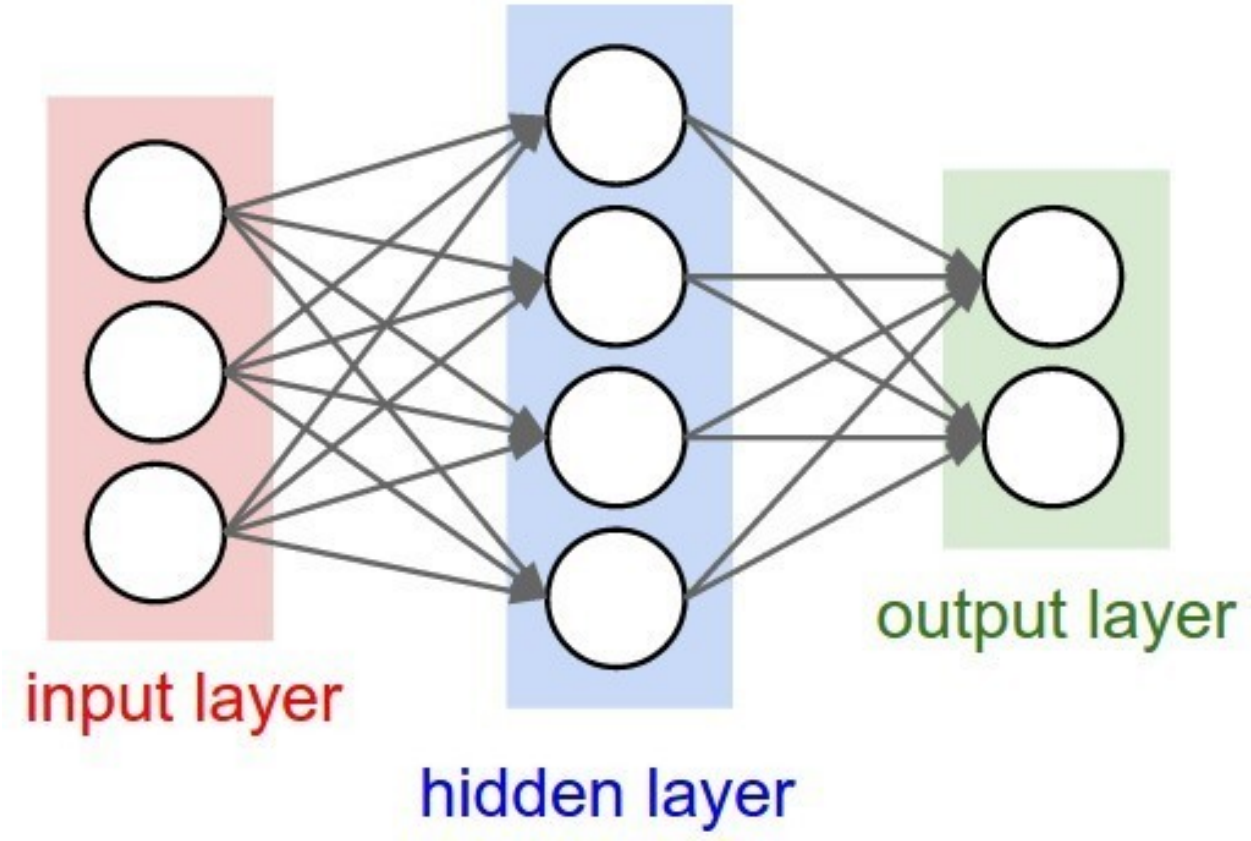


Image credit: CS231n[75]

This simple Neural Network contains:

- Input layer - 3 neurons that should match the size of your input data
- Hidden layer - 4 neurons with weights W that your model should learn during training
- Output layer - 2 neurons that provide the predictions of your model

Want to build a Deep Neural Network? Just add at least one more hidden layer:

[75] https://cs231n.github.io/

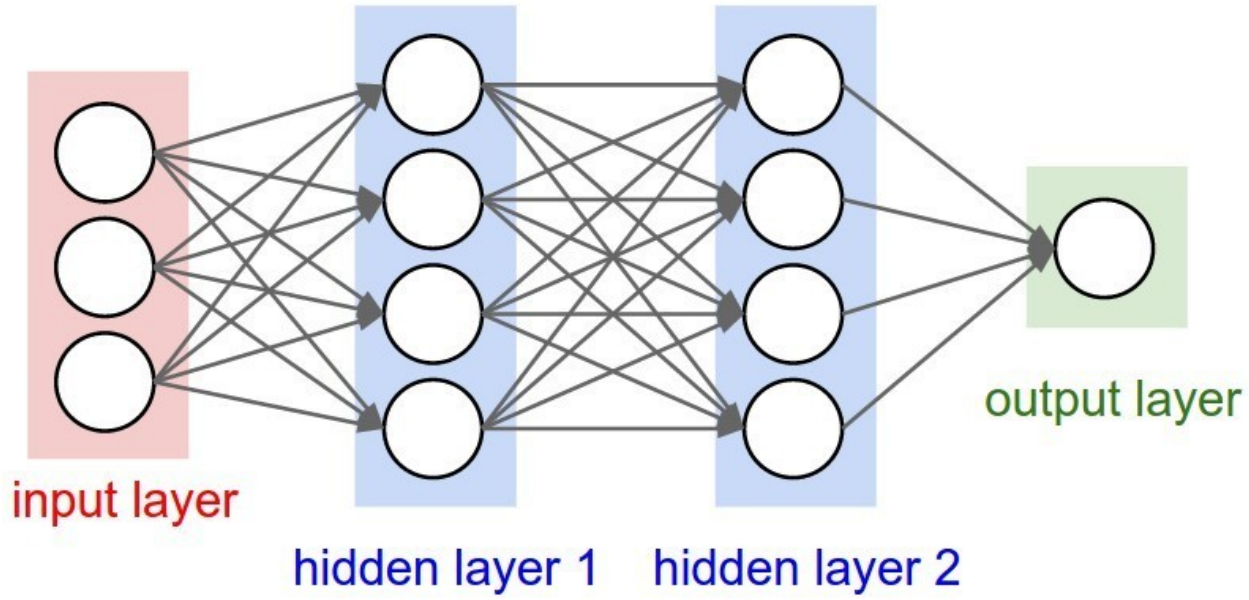
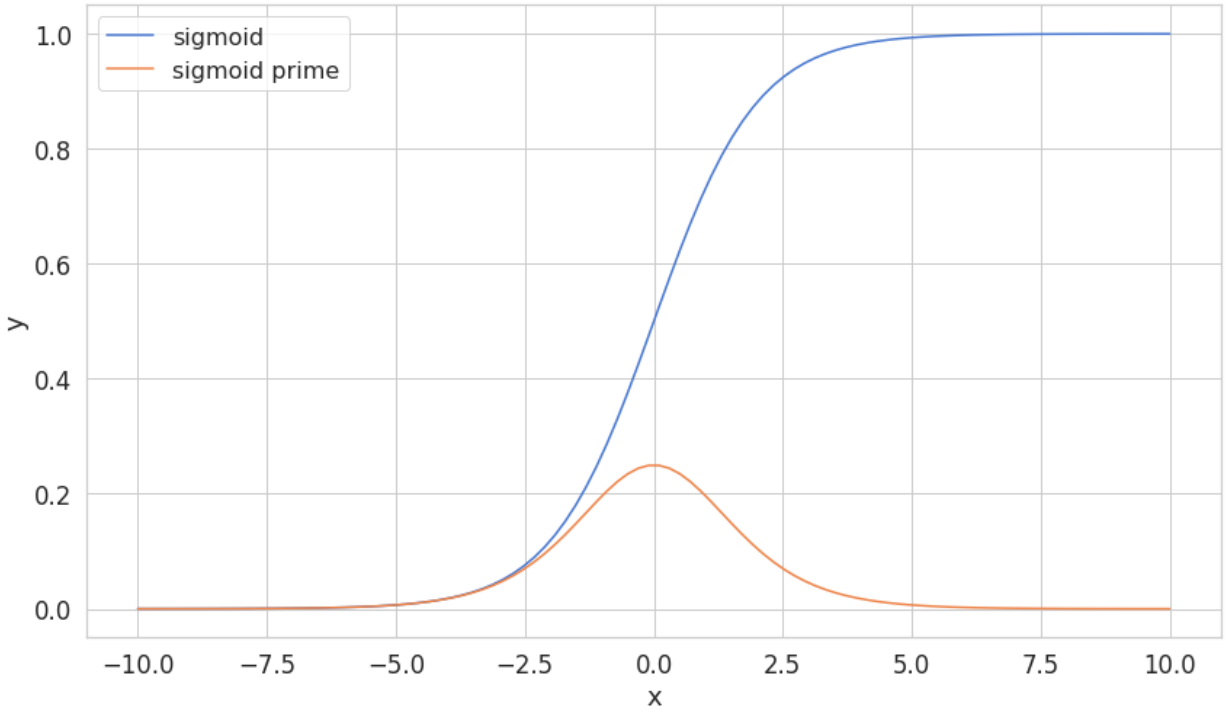


Image credit: CS231n[76]

Sigmoid

The sigmoid function[77] is quite commonly used activation function, at least it was until recently. It has distinct S shape, it is a differentiable real function for any real input value and output values between 0 and 1. Additionally, it has a positive derivative at each point. More importantly, we will use it as an activation function for the hidden layer of our model.

[76] https://cs231n.github.io/
[77] https://en.wikipedia.org/wiki/Sigmoid_function



Here's how it is defined:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Here is how we can implement it:

```
def sigmoid(z):
    return 1.0 / (1.0 + np.exp(-z))
```

It's first derivative (which we will use during the backpropagation step of our training algorithm) has the following formula:

$$\frac{d\sigma(x)}{d(x)} = \sigma(x) \cdot (1 - \sigma(x))$$

Our implementation reuses the sigmoid implementation itself:

```
def sigmoid_prime(z):
    sg = sigmoid(z)
    return sg * (1 - sg)
```

Softmax

The softmax function can be easily differentiated, it is pure (output depends only on input) and the elements of the resulting vector sum to 1. Here it is:

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_K^{k=1} e^{z_k}} \text{for } j = 1, ..., k$$

Here is the Python implementation:

```
def softmax(z):
    return (np.exp(z.T) / np.sum(np.exp(z), axis=1)).T
```

In probability theory, the output of the softmax function is sometimes used as a representation of a categorical distribution. Let's see an example result:

```
softmax(np.array([[2, 4, 6, 8]]))
```

```
array([[ 0.00214401,  0.0158422 ,  0.11705891,  0.86495488]])
```

The output has most of its weight corresponding to the input 8. The softmax function highlights the largest value(s) and suppresses the smaller ones.

Backpropagation

Backpropagation is the backbone of almost anything we do when using Neural Networks. The algorithm consists of 3 subtasks:

1. Make a forward pass
2. Calculate the error
3. Make backward pass (backpropagation)

In the first step, backprop uses the data and the weights of the network to compute a prediction. Next, the error is computed based on the prediction and the provided labels. The final step propagates the error through the network, starting from the final layer. Thus, the weights get updated based on the error, little by little.

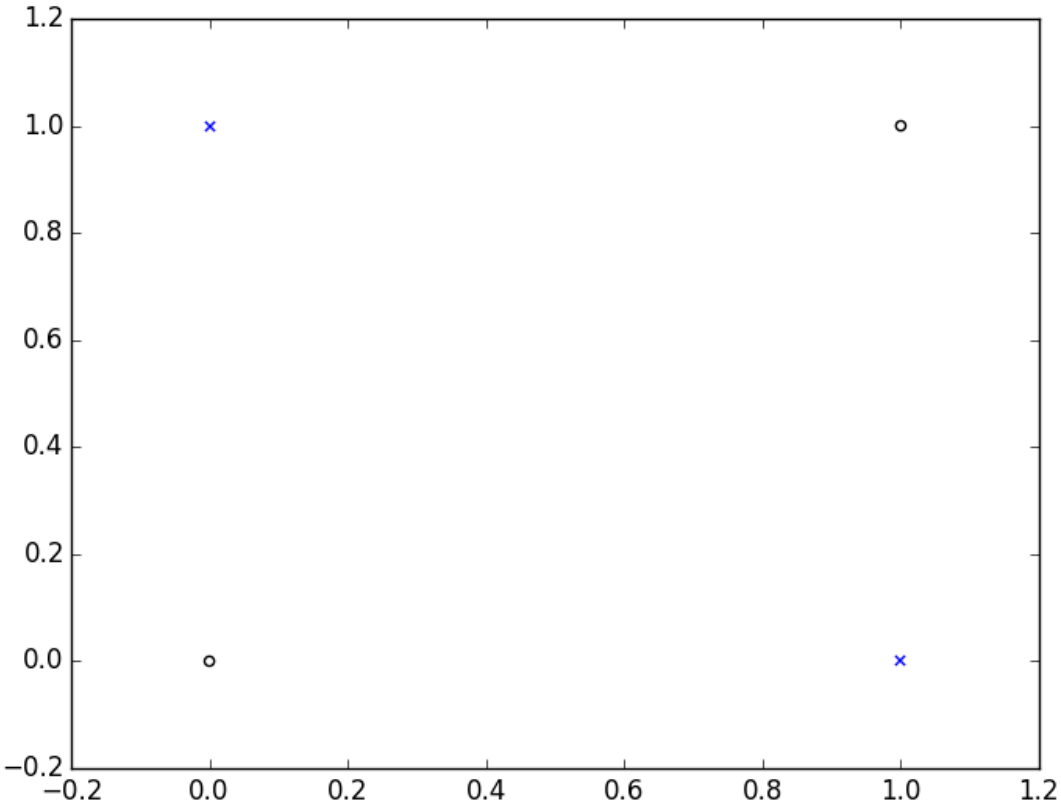
Let's build more intuition about what the algorithm is actually doing:

Solving XOR

We will try to create a Neural Network that can properly predict values from the XOR function. Here is its truth table:

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0

Here is a visual representation:



Let start by defining some parameters:

```
epochs = 50000
input_size, hidden_size, output_size = 2, 3, 1
LR = .1 # learning rate
```

The epochs parameter controls how many times our algorithm will “see” the data during training. Then we set the number of neurons in the input, hidden and output layers - we have 2 numbers as input and 1 number as output size. The learning rate parameter controls how quickly our Neural Network will learn from new data and forget what already knows.

Our training data (from the table) looks like this:

```
X = np.array([[0,0], [0,1], [1,0], [1,1]])
y = np.array([ [0],   [1],   [1],   [0]])
```

The W vectors in our NN need to have some initial values. We'll sample a uniform distribution, initialized with proper size:

```
w_hidden = np.random.uniform(size=(input_size, hidden_size))
w_output = np.random.uniform(size=(hidden_size, output_size))
```

Finally, implementation of the Backprop algorithm:

```
for epoch in range(epochs):

    # Forward
    act_hidden = sigmoid(np.dot(X, w_hidden))
    output = np.dot(act_hidden, w_output)

    # Calculate error
    error = y - output

    if epoch % 5000 == 0:
        print(f'error sum {sum(error)}')

    # Backward
    dZ = error * LR
    w_output += act_hidden.T.dot(dZ)
    dH = dZ.dot(w_output.T) * sigmoid_prime(act_hidden)
    w_hidden += X.T.dot(dH)
```

```
error sum [-1.77496016]
error sum [ 0.00586565]
error sum [ 0.00525699]
error sum [ 0.0003625]
error sum [-0.00064657]
error sum [ 0.00189532]
error sum [  3.79101898e-08]
error sum [  7.47615376e-13]
error sum [  1.40960742e-14]
error sum [  1.49842526e-14]
```

That error seems to be decreasing! YaY! And the implementation is not that scary, isn't it?

During the `forward` step, we take the dot product of the data X and W_{hidden} and apply our activation function to obtain the output of our hidden layer. We obtain the predictions by taking the dot product of the hidden layer output and W_{output}.

To obtain the `error`, we calculate the difference between the true values and the predicted ones. Note that this is a very crude metric, but it works fine for our example.

Finally, we use the calculated error to adjust the weights. Note that we need the results from the forward pass `act_hidden` to calculate W_{output} and calculate the first derivative using `sigmoid_prime` to update W_{hidden}.

In order to make an inference (predictions) we'll do just the forward step (since we won't adjust W based on the result):

```
test_data = X[1] # [0, 1]

act_hidden = sigmoid(np.dot(test_data, w_hidden))
np.dot(act_hidden, w_output)
```

```
array([ 1.])
```

Our sorcery seems to be working! The prediction is correct!

Classifying Images

Building a Neural Network

Our Neural Network will have only 1 hidden layer. We will implement a somewhat more sophisticated version of our training algorithm shown above along with some handy methods.

Initializing the weights

We'll sample a uniform distribution with values between -1 and 1 for our initial weights. Here is the implementation:

```
def _init_weights(self):
    w1 = np.random.uniform(-1.0, 1.0,
                           size=(self.n_hidden_units, self.n_features))
    w2 = np.random.uniform(-1.0, 1.0,
                           size=(self.n_classes, self.n_hidden_units))
    return w1, w2
```

Training

Let's have a look at the training method:

```
def fit(self, X, y):
    self.error_ = []
    X_data, y_data = X.copy(), y.copy()
    y_data_enc = one_hot(y_data, self.n_classes)

    X_mbs = np.array_split(X_data, self.n_batches)
    y_mbs = np.array_split(y_data_enc, self.n_batches)

    for i in range(self.epochs):

        epoch_errors = []

        for Xi, yi in zip(X_mbs, y_mbs):

            # update weights
            error, grad1, grad2 = self._backprop_step(Xi, yi)
            epoch_errors.append(error)
            self.w1 -= self.learning_rate * grad1
            self.w2 -= self.learning_rate * grad2
        self.error_.append(np.mean(epoch_errors))
    return self
```

For each epoch, we apply the backprop algorithm, evaluate the error and the gradient with respect to the weights. We then use the learning rate and gradients to update the weights.

```
def _backprop_step(self, X, y):
    net_input, net_hidden, act_hidden, net_out, act_out = self._forward(X)
    y = y.T

    grad1, grad2 = self._backward(net_input, net_hidden, act_hidden, act_out, y)

    # regularize
    grad1 += self.w1 * (self.l1 + self.l2)
    grad2 += self.w2 * (self.l1 + self.l2)

    error = self._error(y, act_out)

    return error, grad1, grad2
```

Doing a backprop step is a bit more complicated than our XOR example. We do an additional step before returning the gradients - apply L1 and L2 Regularization[78]. Regularization is used to guide our training towards simpler methods by penalizing large values for our parameters W.

```
def _forward(self, X):
    net_input = X.copy()
    net_hidden = self.w1.dot(net_input.T)
    act_hidden = sigmoid(net_hidden)
    net_out = self.w2.dot(act_hidden)
    act_out = sigmoid(net_out)
    return net_input, net_hidden, act_hidden, net_out, act_out

def _backward(self, net_input, net_hidden, act_hidden, act_out, y):
    sigma3 = act_out - y
    sigma2 = self.w2.T.dot(sigma3) * sigmoid_prime(net_hidden)
    grad1 = sigma2.dot(net_input)
    grad2 = sigma3.dot(act_hidden.T)
    return grad1, grad2
```

Our `forward` and `backward` steps are very similar to the one in our previous example, how about the error?

Measuring the error

We're going to use Cross-Entropy loss[79] (known as log loss) function to evaluate the error. This function measures the performance of a classification model whose output is a probability. It penalizes (harshly) predictions that are wrong and confident. Here is the definition:

[78] http://www.chioka.in/differences-between-l1-and-l2-as-loss-function-and-regularization/
[79] https://en.wikipedia.org/wiki/Cross_entropy

$$\text{Cross-Entropy} = -\sum_{c=1}^{C} y_{o,c} \log(p_{o,c})$$

where C is the number of classes, y is a binary indicator if class label is the correct classification for the observation and p is the predicted probability that o is of class c.

The implementation in Python looks like this:

```
def cross_entropy(outputs, y_target):
    return -np.sum(np.log(outputs) * y_target, axis=1)
```

Now that we have our loss function, we can finally define the error for our model:

```
def _error(self, y, output):
    L1_term = L1_reg(self.l1, self.w1, self.w2)
    L2_term = L2_reg(self.l2, self.w1, self.w2)
    error = cross_entropy(output, y) + L1_term + L2_term
    return 0.5 * np.mean(error)
```

After computing the Cross-Entropy loss, we add the regularization terms and calculate the mean error. Here is the implementation for L1 and L2 regularizations:

```
def L2_reg(lambda_, w1, w2):
    return (lambda_ / 2.0) * (np.sum(w1 ** 2) + np.sum(w2 ** 2))

def L1_reg(lambda_, w1, w2):
    return (lambda_ / 2.0) * (np.abs(w1).sum() + np.abs(w2).sum())
```

Making predictions

Now that our model can learn from data, it is time to make predictions on data it hasn't seen before. We're going to implement two methods for prediction - `predict` and `predict_proba`:

```
def predict(self, X):
    Xt = X.copy()
    _, _, _, net_out, _ = self._forward(Xt)
    return mle(net_out.T)
```

Recall that predictions in NN (generally) includes applying a forward step on the data. But the result of it is a vector of values representing how strong the belief for each class is for the data. We'll use Maximum likelihood estimation (MLE)[80] to obtain our final predictions:

[80] https://en.wikipedia.org/wiki/Maximum_likelihood_estimation

```
def mle(y, axis=1):
    return np.argmax(y, axis)
```

MLE works by picking the highest value and return it as a predicted class for the input.

```
def predict_proba(self, X):
    Xt = X.copy()
    _, _, _, _, act_out = self._forward(Xt)
    return softmax(act_out.T)
```

The method `predict_proba` returns a probability distribution over all classes, representing how likely each class is to be correct. Note that we obtain it by applying the `softmax` function to the result of the `forward` step.

Evaluation

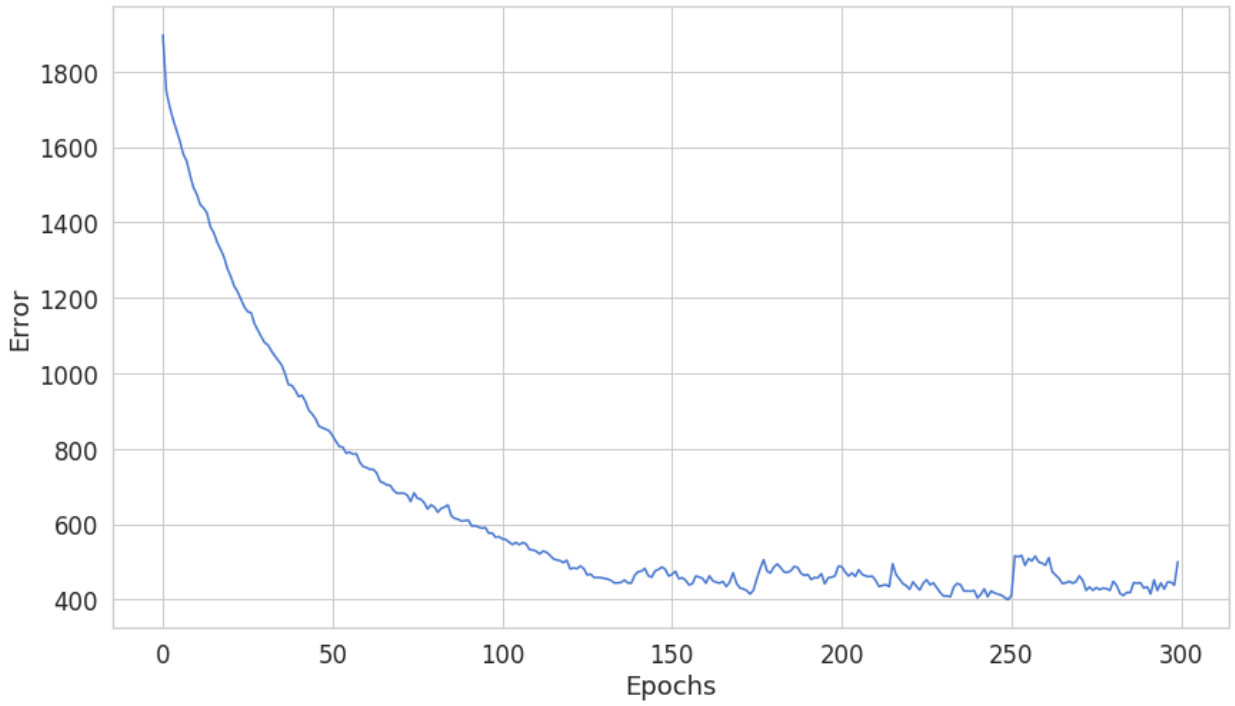
Time to put our NN model to the test. Here's how we can train it:

```
N_FEATURES = 28 * 28 # 28x28 pixels for the images
N_CLASSES = 10

nn = NNClassifier(
    n_classes=N_CLASSES,
    n_features=N_FEATURES,
    n_hidden_units=50,
    l2=0.5,
    l1=0.0,
    epochs=300,
    learning_rate=0.001,
    n_batches=25,
    random_seed=RANDOM_SEED
).fit(X_train, y_train);
```

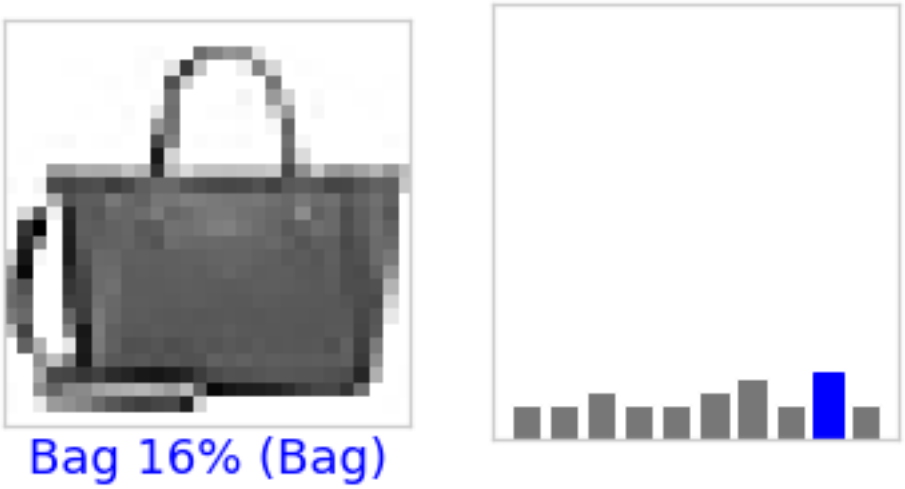
The training might take some time, so please be patient. Let's get the predictions:

```
y_hat = nn.predict_proba(X_test)
```

First, let's have a look at the training error:



Something looks fishy here, seems like our model can't continue to reduce the error 150 epochs or so. Let's have a look at a single prediction:



That one seems correct! Let's have a look at few more:

Not too good. How about the training & testing accuracy:

```
print('Train Accuracy: %.2f%%' % (nn.score(X_train, y_train) * 100))
print('Test Accuracy: %.2f%%' % (nn.score(X_test, y_test) * 100))
```

```
Train Accuracy: 50.13%
Test Accuracy: 49.83%
```

Well, those don't look that good. While a random classifier will return ~10% accuracy, ~50% accuracy on the test dataset will not make a practical classifier either.

Improving the accuracy

That "jagged" line on the training error chart shows the inability of our model to converge. Recall that we use the Backpropagation algorithm to train our model. Training Neural Nets converge much

faster when data is normalized[81].

We'll use scikit-learn's scale[82] to normalize our data. The documentation states that:

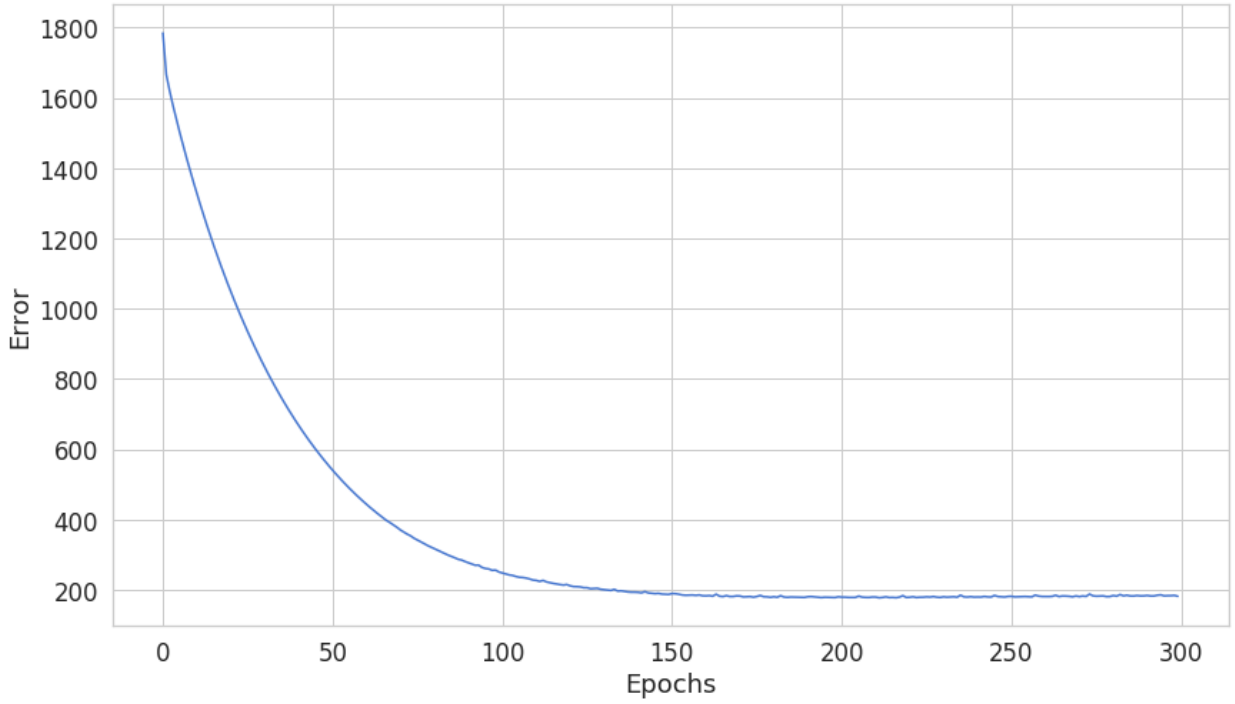
> Center to the mean and component wise scale to unit variance.

Here is the new training method:

```
from sklearn.preprocessing import scale

X_train_scaled = scale(X_train.astype(np.float64))
X_test_scaled = scale(X_test.astype(np.float64))

nn = NNClassifier(
    n_classes=N_CLASSES,
    n_features=N_FEATURES,
    n_hidden_units=50,
    l2=0.5,
    l1=0.0,
    epochs=300,
    learning_rate=0.001,
    n_batches=25,
    random_seed=RANDOM_SEED
).fit(X_train_scaled, y_train);
```

Let's have a look at the error:

[81] https://arxiv.org/abs/1502.03167
[82] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.scale.html



The error seems a lot more stable and settles at lower point - ~200 vs ~400. Let's have a look at some predictions:

Those look much better, too! Finally, the accuracy:

```
print('Train Accuracy: %.2f%%' % (nn.score(X_train_scaled, y_train) * 100))
print('Test Accuracy: %.2f%%' % (nn.score(X_test_scaled, y_test) * 100))
```

```
Train Accuracy: 92.13%
Test Accuracy: 87.03%
```

~87% (vs ~50%) on the training set is a vast improvement over the unscaled method. Finally, your hard work paid off!

Conclusion

What a ride! I hope you got a blast working on your first Neural Network from scratch, too!

You learned how to process image data, transform it, and use it to train your Neural Network. We used some handy tricks (scaling) to vastly improve the performance of the classifier.

Complete source code in Google Colaboratory Notebook[83]

[83] https://drive.google.com/file/d/1S59KWV8KmZTI-A6OpXge3yG87HXyIcUz/view?usp=sharing

Build a taxi driving agent in a post-apocalyptic world using Reinforcement Learning

> TL;DR Build a simple MDP for self-driving taxi problem. Pick up passengers, avoid danger and drop them off at specified location. Build an agent and solve the problem using Q-learning.

You wake up. It is a sunny day. Your partner is still asleep next to you. You take a minute to admire the beauty and even crack a smile.

Your stomach is rumbling, so you jump on your feet and look around. Jackpot! You're looking at the leftovers of the anniversary dinner you two had last night. You take a minute to scratch some private spot of yours. Yep, what an amazing morning! You have 1/3 of a bean can, expired only 6 months ago. It is delicious!

Although you feel bad about not leaving any food for the one sleeping in your bed, you dress up and prepare for work. "What are you going to do? Not eat and work!?" - those thoughts don't seem to help anymore. Time to hop in the car!

You try to reach Johny via the radio but no luck. Oh well, Jimmy is missing since last week, his twin brother might be too. Strange, you can't remember the last time your eyes were feeling wet.

A lot has changed since *the event*. The streets are getting more dangerous every week, but people still need transportation. You receive a pickup request and have a look at the anomaly map. You accept hesitantly. Your map was updated 2 months ago.

You got the job done and got credit for a half can. That couple was really generous! That leaves time for tinkering with the idea of making self-driving taxis. You even got a name - *SafeCab*. You read a lot of about this crazy guy called Elon Nusk that was trying to build those fully autonomous vehicles right before *the event* occured.

You start scratching your head. Is this possible or just a fantasy? After all, if you get this done, you might afford to eat every other day. Enter Reinforcement Learning.

Complete source code in Google Colaboratory Notebook[84]

[84] https://colab.research.google.com/drive/1FMo6Lpf1UtO1blfMyA4yznzDN7tlgIWm

Reinforcement Learning

Reinforcement Learning (RL) is concerned with producing algorithms (agents) that try to achieve some predefined goal. The achievement of this objective is dependent on choosing a set of actions - receiving rewards for the good ones and punishment for the bad ones - the reinforcement bit comes from this. The agent act, in environments that have a state, gives rewards, and a set of actions.

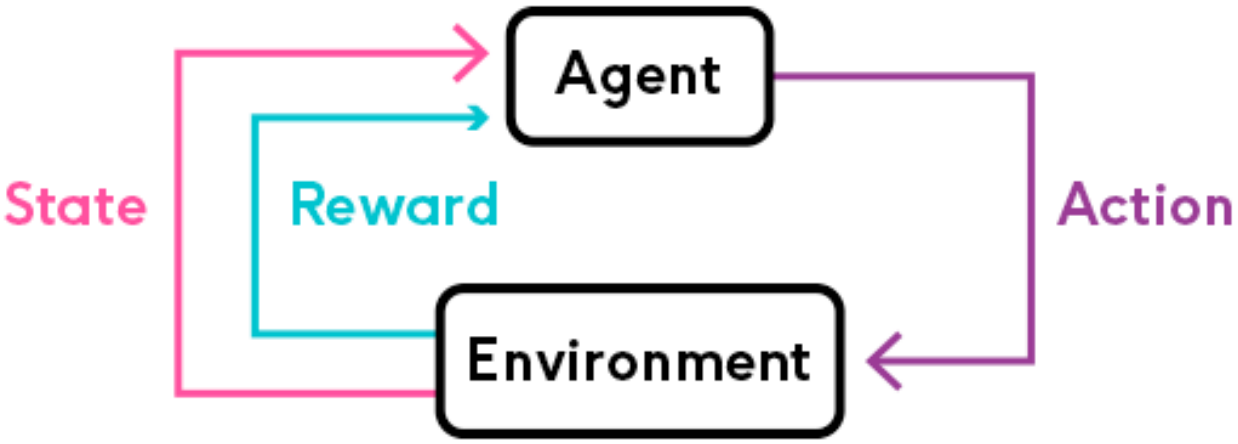


Image credit: https://phrasee.co/[85]

Deep Reinforcement Learning (using Deep Neural Networks for choosing actions) achieved some great things lately:

- Play Atari games[86]
- Play Doom[87]
- Dominate the game of Go[88]

Markov Decision Processes

Markov Decision Process (MDP) is a mathematical formulation of the RL problem. MDPs satisfy the Markov property:

Markov property - the current state completely represents the state of the environment (world). That is, the future depends only on the present.

An MDP can be defined by (S, A, R, P, γ) where:

[85] https://phrasee.co/
[86] https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf
[87] https://www.youtube.com/watch?v=oo0TraGu6QY
[88] https://deepmind.com/blog/alphago-zero-learning-scratch/

- S - set of possible states
- A - set of possible actions
- R - probability distribution of reward given (state, action) pair
- P - probability distribution over how likely any of the states is to be the new states, given (state, action) pair. Also known as transition probability.
- γ - reward discount factor

The discount factor γ allows us to inject the heuristic that 100 bucks now are more valuable than 100 bucks in 30 days. A discount factor of 1 would make future rewards worth just as much as immediate rewards.

Another reason to discount future rewards:

> In the long run we are all dead - J. M. Keynes

Learning

Here is how learning happens in RL context:

for time step $t = 0$ until done:

1. The environment gives your agent a state
2. Your agent chooses an action (from a set of possible ones)
3. The environment gives a reward along with a new state
4. Continue until the goal or other condition is met

What is the objective of all this? Find a function π^*, known as optimal policy, that maximizes the cumulative discounted reward:

$$\sum_{t \geq 0} \gamma^t r_t$$

where r_t is the reward received at step t and γ^t is the discount factor at step t.

A policy π is a function that maps state s to action a, that our agent believes is the best given that state.

Value function

The value function gives you the maximum expected future reward the agent will get, starting from some state s and following some policy π.

$$V_\pi(s) = \underset{\pi}{\mathbb{E}}[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s]$$

There exists an **optimal value function** that has the highest value for all states. Defined as:

$$V^*(s) = \max_\pi V^\pi(s) \quad \forall s \in \mathbb{S}$$

Q-function

Similarly, let's define another function known as Q-function (state-action value function) that gives the expected return starting from state s, taking action a, and following policy π.

$$Q_\pi(s, a) = \mathop{\mathbb{E}}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a]$$

You can think of the Q-function as the "quality" of a certain action given a state. We can define the optimal Q-function as:

$$Q^*(s, a) = \max_\pi Q^\pi(s, a) \quad \forall s \in \mathbb{S}$$

There is a relationship between the two optimal functions V^* and Q^*. It is given by:

$$V^*(s) = \max_a Q^*(s, a) \quad \forall s \in \mathbb{S}$$

That is, the maximum expected total reward when starting at s is the maximum of $Q^*(s, a)$ over all possible actions.

We can find the optimal policy π^* by choosing the action a that gives maximum reward $Q^*(s, a)$ for state s:

$$\pi^*(s) = \arg\max_a Q^*(s, a) \quad \forall s \in \mathbb{S}$$

There seems to be a synergy between all functions we defined so far. More importantly, we can now build an optimal agent for a given environment. Can we do it in practice?

Q-learning

Q-Learning is an off-policy algorithm for Temporal Difference (TD) learning[89]. It is proven that with enough training, it converges with probability 1 to a close approximation of the action-value function for an arbitrary target policy. It learns the optimal policy even when actions are selected using exploratory (some randomness) policy (off-policy).

Given a state s and action a, we can express $Q(s, a)$ in terms of itself (recursively):

[89] https://en.wikipedia.org/wiki/Temporal_difference_learning

$$Q(s, a) = r + \gamma \max_{a'} Q(s', a')$$

This is the **Bellman equation**. It defines the maximum future reward as the reward r the agent received, for being at state s, plus the maximum future reward for state s' for every possible action.

We can define Q-learning as an iterative approximation of Q^* using the Bellman equation:

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t))$$

where α is the learning rate that controls how much the difference between previous and new Q value is considered.

Here's the general algorithm we're going to implement:

1. Initialize a Q-values table
2. Observe initial state s
3. Choose action a and act
4. Observe reward r and a new state s'
5. Update the Q table using r and the maximum possible reward from s'
6. Set the current state to the new state and repeat from *step 2* until a terminal state

Note that Q-learning is just one possible algorithm to solve the RL problem. Here is a comparison between some more of them[90].

Exploration vs exploitation

You might've noticed that we glanced over the strategy on choosing an action. Any strategy you implement will have to choose how often to try something new or use something it already knows. This is known as the exploration/exploitation tradeoff.

- Exploration - finding new information about the environment
- Exploitation - using existing information to maximize the reward

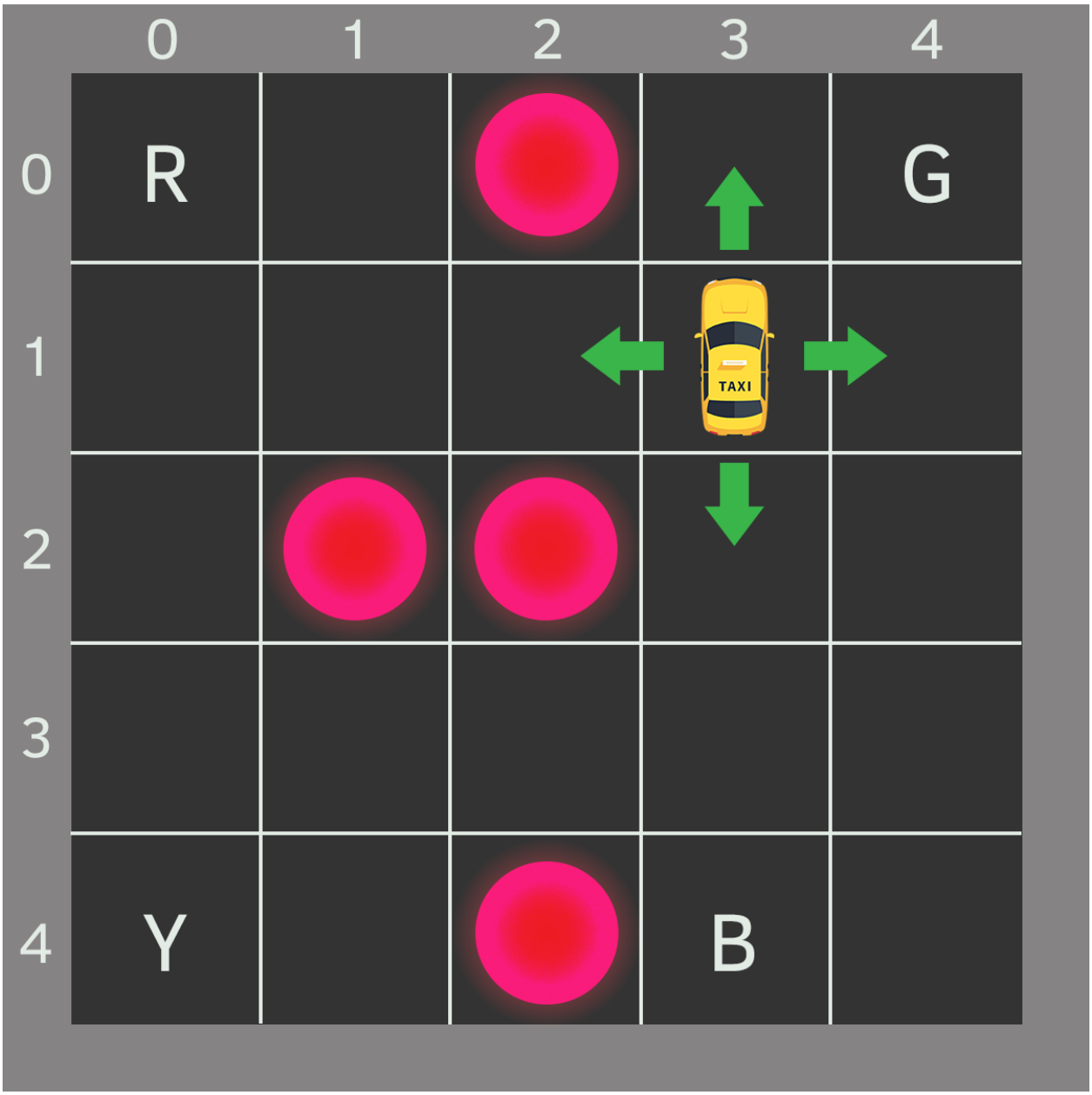
Remember, the goal of our RL agent is to maximize the expected cumulative reward. What does this mean for our self-driving taxi?

Initially, our driving agent knows pretty much nothing about the best set of driving directions for picking up and dropping off passengers. It should learn to avoid anomalies too since they are bad for business (passengers tend to disappear or worse in those)! During this time, we expect to make a lot of exploration.

After obtaining some experience, the agent can use it to choose an action more and more often. Eventually, all choices will be based on what is learned.

[90] https://en.wikipedia.org/wiki/Reinforcement_learning#Comparison_of_reinforcement_learning_algorithms

Driving in a post-apocalyptic world



Your partner sketched this map. Each block represents a small region of the city. Anomalies are marked in bright circles. The four letters are "safe zones", where pickups and drop-offs happen.

Let's assume your car is the only vehicle in the city (not much of a stretch). We can break it up into a 5x5 grid, which gives us 25 possible taxi locations.

The current taxi location is `(3, 1)` in (row, col) coordinates. The pickup and drop off locations are: `[(0,0), (0,4), (4,0), (4,3)]`. Anomalies are at `[(0, 2), (2, 1), (2, 2), (4, 2)]`.

Environment

We're going to encode our city map into an environment for the self-driving agent using OpenAI's Gym[91]. What is this Gym thing anyways?

> Gym is a toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents everything from walking[92] to playing games like Pong[93] or Pinball[94].

The most important entity in Gym is the Environment. It provides a unified and easy to use interface. Here are the most important methods:

- `reset` - resets the environment and returns a random initial state
- `step(action)` - take action and advance one timestep. It returns:

  observation - the new state of the environment

  reward - reward received from taking the action

  done - most environments are divided into well-defined episodes and if `done` is `True` it indicates the episode has been completed

  info - additional information about the environment (might be useful for debugging)
- `render` - renders one frame of the environment (useful for visualization)

The complete source code of the environment is in the notebook. Here, we'll take a look at the registration to Gym:

```
register(
    id='SafeCab-v0',
    entry_point=f"{__name__}:SafeCab",
    timestep_limit=100,
)
```

Note that we set a `timestep_limit` which limits the number of steps in an episode.

Action Space

Our agent encounters one of the 500 states (5 rows x 5 columns x 5 passenger locations x 4 destinations) and it chooses an action. Here are the possible actions:

- south

[91] https://gym.openai.com/
[92] https://gym.openai.com/envs/Humanoid-v2/
[93] https://gym.openai.com/envs/Pong-ram-v0/
[94] https://gym.openai.com/envs/VideoPinball-ram-v0/

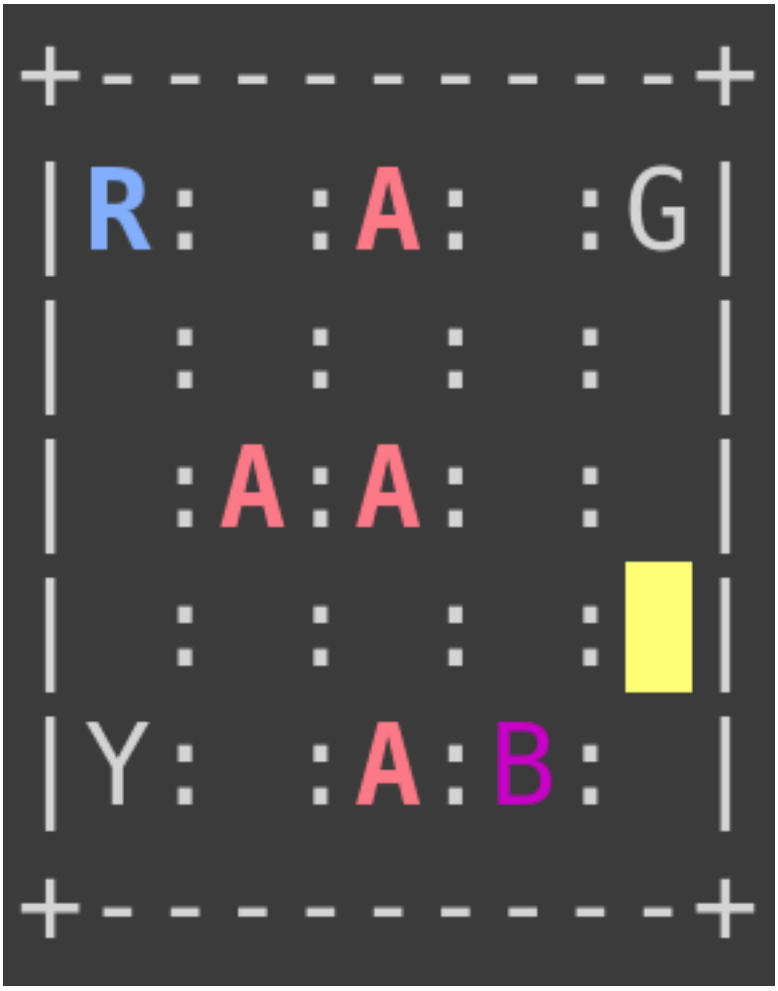
- north
- east
- west
- pickup
- dropoff

You might notice that in the illustration above, the taxi cannot perform certain actions when near the boundaries of the city. We will penalize this with -1 and won't move the taxi if such situation occurs.

Getting a taste of the environment

Let's have a look at our encoded environment:

```
env.reset()
env.render()

print("Action Space {}".format(env.action_space))
print("State Space {}".format(env.observation_space))
```




```
Action Space Discrete(6)
State Space Discrete(500)
```

Here is the action to number mapping:

- 0 = south
- 1 = north
- 2 = east
- 3 = west
- 4 = pickup
- 5 = dropoff

Building an agent

Our agent has a simple interface for interacting with the environment. Here it is:

```
class Agent:

  def __init__(
    self,
    n_states,
    n_actions,
    decay_rate=0.0001,
    learning_rate=0.7,
    gamma=0.618
  ):
    pass

  def choose_action(self, explore=True):
    pass

  def learn(
    self,
    state,
    action,
    reward,
    next_state,
    done,
    episode
  ):
    pass
```

We're going to use the `choose_action` method when we want our agent to make a decision and act. Then, after the reward and new state from the environment are observed, our agent will learn from its actions using the `learn` method.

Let's take a look at the implementations:

```
def __init__(
  self,
  n_states,
  n_actions,
  decay_rate=0.0001,
  learning_rate=0.7,
  gamma=0.618
):
  self.n_actions = n_actions
  self.q_table = np.zeros((n_states, n_actions))
  self.max_epsilon = 1.0
```

```
    self.min_epsilon = 0.01
    self.epsilon = self.max_epsilon
    self.decay_rate = decay_rate
    self.learning_rate = learning_rate
    self.gamma = gamma # discount rate
    self.epsilons_ = []
```

The interesting part of the `__init__` is the initialization of our Q-table. Initially, it is all zeros. Can we initialize it in some other way?

```
def choose_action(self, explore=True):
  exploration_tradeoff = np.random.uniform(0, 1)

  if explore and exploration_tradeoff < self.epsilon:
    # exploration
    return np.random.randint(self.n_actions)
  else:
    # exploitation (taking the biggest Q value for this state)
    return np.argmax(self.q_table[state, :])
```

Our strategy is rather simple. We draw a random number from a uniform distribution between 0 and 1. If this number is smaller than epsilon and we want to explore, we take a random action. Otherwise, we take the best action based on our current knowledge.

```
def learn(
  self,
  state,
  action,
  reward,
  next_state,
  done,
  episode
):
  # Update Q(s,a):= Q(s,a) + lr [R(s,a) + gamma * max Q(s',a') - Q(s,a)]
  self.q_table[state, action] = self.q_table[state, action] + \
    self.learning_rate * (reward + self.gamma * \
      np.max(self.q_table[new_state, :]) - self.q_table[state, action])

  if done:
    # Reduce epsilon to decrease the exploration over time
    self.epsilon = self.min_epsilon + (self.max_epsilon - self.min_epsilon) * \
      np.exp(-self.decay_rate * episode)
    self.epsilons_.append(self.epsilon)
```

Learning involves the update of the Q-table using the Q-learning equation and reducing the exploration rate ϵ if the episode is complete.

Training

Now that our agent is ready for action, we can train it on the environment we've created. Let's train our agent for 50k episodes and record episode rewards over time:

```
total_episodes = 60000
total_test_episodes = 10

agent = Agent(env.observation_space.n, env.action_space.n)
```

Let's have a look at our agent driving **before learning anything about the environment**:

See YouTube video[95]

```
rewards = []

for episode in range(total_episodes):
    state = env.reset()
    episode_rewards = []

    while True:

        action = agent.choose_action()

        # Take the action (a) and observe the outcome state(s') and reward (r)
        new_state, reward, done, info = env.step(action)

        agent.learn(
          state,
          action,
          reward,
          new_state,
          done,
          episode
        )

        state = new_state

```

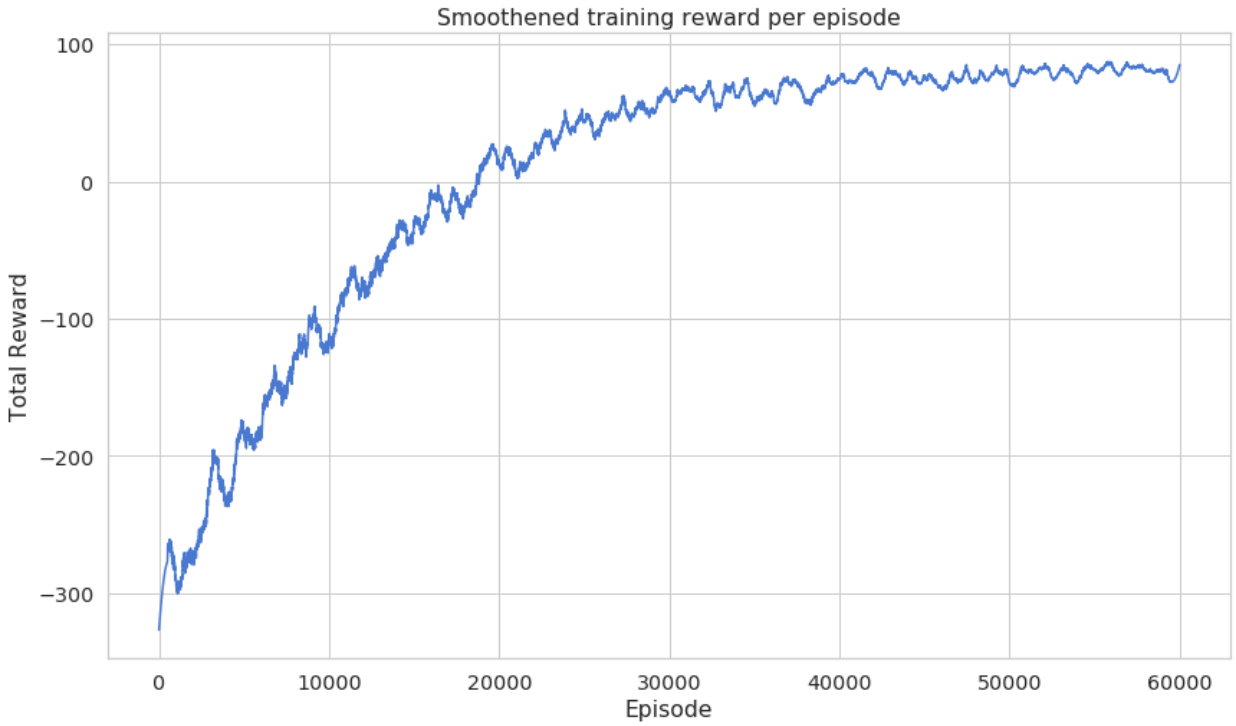
[95] https://www.youtube.com/watch?v=CHayUf_IoUc

```
25          episode_rewards.append(reward)
26
27          if done == True:
28            break
29
30      rewards.append(np.mean(episode_rewards))
```

Recall that we set `timestep_limit` when we registered the environment so our agent won't stay in an episode infinitely.

Evaluation

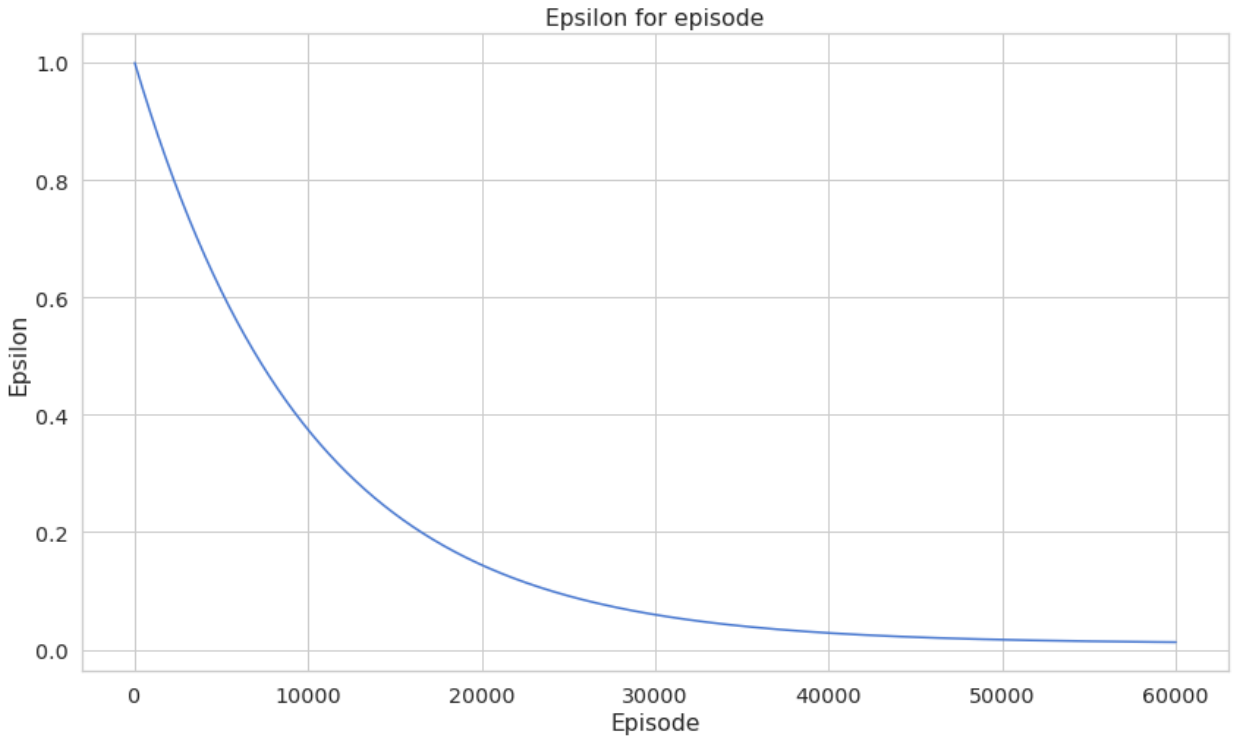
Let's have a look at reward changes as the training progresses:



Note that the learning curve is smoothened using savgol_filter[96] `savgol_filter(rewards, window_length=1001, polyorder=2)`

Recall that our exploration rate should decrease as our agent is learning. Have a look:

[96] https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.savgol_filter.html



Here's how we're going to test our agent and record the progress:

```
frames = []

rewards = []

for episode in range(total_test_episodes):
    state = env.reset()
    episode_rewards = []

    step = 1

    while True:
        action = agent.choose_action(explore=False)

        new_state, reward, done, info = env.step(action)

        frames.append({
          'frame': env.render(mode='ansi'),
          'state': state,
          'episode': episode + 1,
          'step': step,
          'reward': reward
```

```
        })

        episode_rewards.append(reward)

        if done:
            step = 0
            break
        state = new_state
        step += 1

    rewards.append(np.mean(episode_rewards))

env.close()
```

Note that we want our agent to use only the experience it has so we set `explore=False`. Here's what the total reward for each episode looks like:

I know that this chart might not give you a good idea of what the agent is capable of. Here is a video of it driving in the city:

See YouTube video[97]

Pretty good, right? It looks like that it learned to avoid the anomalies, pick up and drop off passengers.

[97] https://www.youtube.com/watch?v=HHIuVNZNMNg

Conclusion

Congratulations on building a self-driving taxi agent. You've learned how to

- Build your own environment based on one provided by OpenAI's Gym
- Implement and apply Q-learning
- Build an agent that learns to pick up, drop off passengers and avoid danger areas

Complete source code in Google Colaboratory Notebook[98]

Can you increase the size of the city? Does the agent still learn well? Tell me how it went in the comments below!

[98] https://colab.research.google.com/drive/1FMo6Lpf1UtO1blfMyA4yznzDN7tlgIWm