K-means

A clustering algorithm looks at a number of data points and automatically finds data points that are related or similar to each other. K-means is a popular clustering algorithm used in machine learning and data analysis. The algorithm follows these steps:

- **Initializing** : Choose K initial centroids from the training set randomly.
- **Assignment** : Assign each data point to the cluster whose centroid is closest to it. This is typically done using a distance metric such as Euclidean distance.
- **Update Centroids** : Recalculate each centroid by taking the mean of all the data points assigned to that cluster.
- Keep looping until convergence. You can manually set the times of iteration.

So what is K-means doing? Actually, K-means is trying to optimize the following cost function(distortion). Assume that $c^{(i)}$ is the cluster that the example $x^{(i)}$ is belong to. μ_i is the centroid of cluster i:

$J(c, \mu) = \frac{1}{m} \sum_{i=1}^{m} ||x^{(i)} - \mu_{c^{(i)}}||^2$

Because the new centroid is the center of all data points assigned to that cluster, we reduce the distortion as much as possible.

But since the algorithm is likely to be stuck in local minimum, we can randomly choose the initial centroids for $[50, 1000]$ times and get the solution that has the minimum distortion.

Last but not least, let's discuss our viewpoint on how to choose K. Often, you want to get clusters for some later purpose. Evaluate K-means based on how well it performs on that later purpose.

For example, if we want to do clustering on T-shirt, choose $K = 3$ so that we can divide them into small, medium, and large T-shirts.

Anomaly Detection

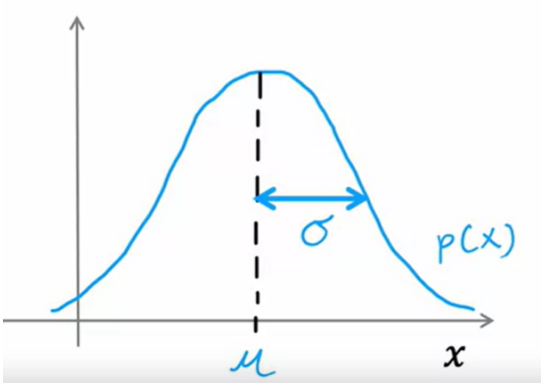
Algorithm

Anomaly detection algorithms look at an unlabeled dataset of normal events and learns to detect anomalous event. When you're given your training sets of these m examples, the first thing you do is to build a model for the probability of x.

In order to apply anomaly detection, we are going to use Gaussian distribution (normal distribution). Say x is a number. Probability of x is determined by a Gaussian with mean μ, variance σ^2:

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$$

And if we plot the curve on the graph:



Given a training set $\{x^{(1)}, x^{(2)} \dots x^{(m)}\}$, it's easy for us to fit a Gaussian distribution:

$$\mu = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}$$

$$\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} (x^{(i)} - \mu)^2$$

We can do anomaly detection for univariable. But for many practical applications you might do this with dozens or even hundreds of features. We can calculate the probability like this:

$$p(\vec{x}) = \prod_{i=1}^{n} p(x_i)$$

You may recognize that this equation corresponds to assuming that the features are statistically independent. But in practice the algorithm works fine even if the features are not independent.

Evaluation

If you can have a way to evaluate a system, it you will be able to make decisions more quickly.

Assume we have some labeled data of anomalous ($y = 1$) and non-anomalous ($y = 0$) examples.

For example, if we are monitoring aircraft engines. There are overall 10000 good engines (normal) and 20 flawed engines (anomalous).

Training set	**6000 good engines**	
CV	**2000 good engines**	**10 anomalous**
Test	**2000 good engines**	**10 anomalous**

If the labeled anomalous examples are too small, we can use no test set and get more anomalous examples in cross-validation set.

When evaluating the detection system, we can apply F1-score since the samples is a skewed dataset. First we can calculate precision and recall. The F1-score is:

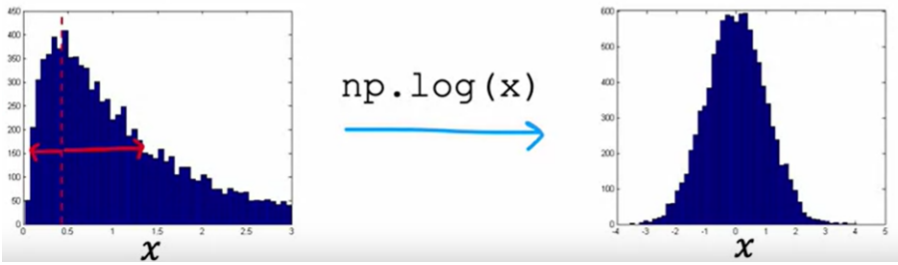
$F_1\ score = 2 \cdot \frac{precision \times recall}{precision + recall}$

Anomaly detection vs. supervised learning

Anomaly detection	supervised learning
Very small number of positive examples. Large number of negative examples.	Large number of positive and negative examples.
Many different types of anomalies. Hard for algorithm to learn what the anomalies look like.	Enough positive examples for algorithm to get a sense of what positive examples are like.
Future anomalies may look nothing like any of the anomalous examples we've seen so far.	Future positive examples are likely to be similar to ones in training set.

Choose what feature to use

First, in order to make our distribution look more Gaussian, we can apply some methods that are used in feature engineering. For example, you may replace the feature x with $\log x$:



What's more, we can choose the features that might take on unusually large or small values in the event of an anomaly.

For instance, image that we are monitoring computers in a data center. We measure CPU load(high) and network traffic(low). It seems nothing happens. But if we add a feature:

$$x' = \frac{\text{CPU load}}{\text{network traffic}}$$

It turns out that for computers that have been hacked, the feature x' is very high. So that adding some feature with feature engineering techniques will be more efficient.

Collaborative filtering

Use per-item features

Say you run a large movie streaming website and your users have rated movies using one to five stars:

Movie	Alice(1)	Bob(2)	Carol(3)	Dave(4)
Love at last	5	5	0	0
Romance forever	5	?	?	0
Cute puppies of love	?	4	0	?
Nonstop car chases	0	0	5	4
Swords vs. karate	0	0	5	?

How can we predict the rating that user will give to the movies they haven't seen? As we all know, user give rating to the movies according to the feature the movies have and how do they like the feature.

For example, the movie *Love at last* is a very romantic movie, so this feature takes on 1, but it's not at all an action movie. So this feature takes on 0. We use $\vec{w} = (1, 0)$ to represent the feature of the movie.

Suggest that Alice has a strong interest in romantic movie and do not like action movie at all. So we set $\vec{x}_1 = (5, 0)$ in order to quantify how Alice like these features. And for Carol, may she like action movies very much. So we set $x_3 = (0, 5)$.

It turns out that $\vec{w} \cdot \vec{x}_1 = 5$ and $\vec{w} \cdot \vec{x}_2 =$0. In conclusion, we can predict these features and make recommendations accordingly.

Collaborative filtering

Before we start out discussion, Let's give some notations to the variables related to the problem.

Name	Description
$r(i, j)$	scalar; 1 if user i has rated movie j, 0 otherwise.
$y(i, j)$	scalar; rating given by user j on movie i
$\mathbf{w}^{(j)}$	vector; parameter for user j
$b^{(j)}$	scalar; bias for use j
$\mathbf{x}^{(i)}$	vector; parameter for movie i
n	scalar; the number of movies
m	scalar; the number of users
k	scalar; the number of features

The collaborative filtering assumes that if two users have similar preference on certain items, they will have similar preference to other items as well. Its cost function is given by:

$$J(\mathbf{X}, \mathbf{b}, \mathbf{W}) = \frac{1}{2} \sum_{i,j} r(i, j) \times (w^{(j)} \cdot x^{(i)} + b^{(j)} - y(i, j))^2 + \text{regularization}$$

where:

$$\text{regularization} = \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{k} x(i, j)^2 + \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{k} w(i, j)^2$$

Observations are that the cost function is very similar to linear regression. So we can apply gradient descent to the collaborative filtering in order to give a better estimate of the parameters.

Binary labels (optional)

The collaborative filtering also works when it comes to binary labels. Define $g(z) = \frac{1}{1+e^{-z}}$ and $f(x) = g(w^{(j)} \cdot x^{(i)} + b^{(j)})$. The cost function is:

$$J(\mathbf{X}, \mathbf{b}, \mathbf{W}) = \sum_{i,j} r(i, j) \times L(f(x), y(i, j))$$

where:

$$L(f(x), y(i,j)) = -y(i,j) \cdot \log[f(x)] - (1 - y(i,j)) \cdot \log[1 - f(x)]$$

TensorFlow implementation

We'll take a look at how you can use TensorFlow to implement the collaborative filtering algorithm.

`tf.GradientTape()` can automatically figure out for you what are the derivatives of the cost function:

```
# generate the initial parameter with the normal
distribution
W = tf.Variable(tf.random.normal(shape = (num_users,
num_features), dtype = tf.float64))
X = tf.Variable(tf.random.normal(shape = (num_movies,
num_features), dtype = tf.float64))
b = tf.Variable(tf.random.normal(shape = (1, num_users),
dtype = tf.float64))
optimizer = tf.keras.optimizers.Adam(learning_rate = 0.1)

iterations = 200
lam = 1 # regularization
for i in range(iterations):
    # tf.GradientTape is a context manager
    # so we use "with...as" syntax
    with tf.GradientTape() as tape:
        # use vectorization to speed up (it is very
fast!)
        cost_value = cofi_vectorization(X, W, b, Ynorm,
R, lam)
    # compute gradient
    grads = tape.gradient(cost_value, [X, W, b])
    # grads is a list (partial derivative of X, W and b)
    optimizer.apply_gradients(zip(grads, [X, W, b]))
```

Content-based filtering

Content-based filtering

To get started, let's compare collaborative filtering approach with this new content-based filtering approach. Let's take a look:

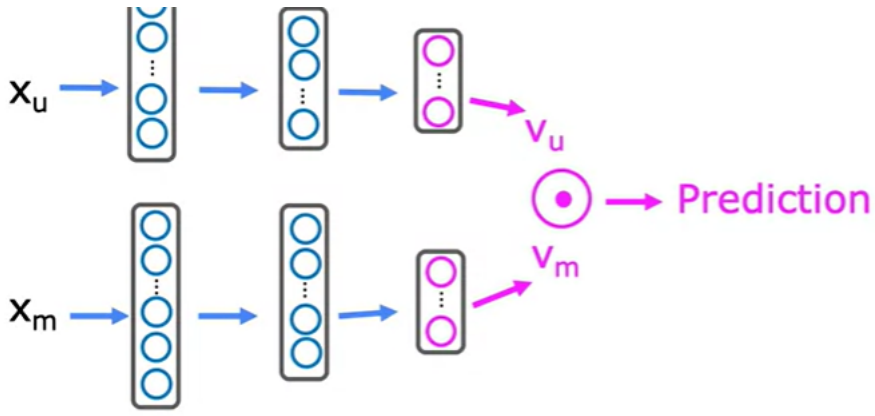
- Collaborative filtering: Recommend items to you based on ratings of users who gave similar ratings as you.
- Content-based filtering: Recommend items to you based on features of user and item to find good match.

Suppose $x_u^{(j)}$ is the profile of users and $x_m^{(i)}$ is the features of items. We can compute a vector $v_u^{(j)}$ from $x_u^{(j)}$ and another vector $v_m^{(i)}$ from $x_m^{(i)}$.

If we are able to come up with an appropriate choice of these vectors, then hopefully the dot product between these two vectors will be a good prediction of the rating that user j gives movie i.

A good way to develop a content-based filtering algorithm is to use deep learning. The first neural network will be user network, whose input is x_u and output is v_u. The second neural network will be movie network, whose input is x_m and output is v_m

Then we add a dot layer to compute the dot product of v_u and v_m. So the architecture of neural network looks like this:



After building the neural network, we can find movies similar to movie i just by computing:

$\min_k ||v_m^{(k)} - v_m^{(i)}||^2$

The Functional API

The Keras functional API is a way to create models that are more flexible that the `keras.sequentail` API. The main idea is that a deep learning model is usually a directed acyclic graph (DAG) of layers. So the functional API tries to build graph of layers.

To build this model using the functional API, start by creating an input node:

```
inputs = keras.Input(shape = (784, ))
```

The returned tensor is a symbolic placeholder that will be used as the starting point for building the neural network architecture. It doesn't contain any actual data. Instead, this tensor will be connected to subsequent layers in the model, defining how the data flows through the network.

You create a new node in the graph of layers by calling a layer on this inputs object:

```
dense = layers.Dense(64, activation = "relu")
x = dense(inputs) # "layer call"
```

The "layer call" action is like drawing an arrow from "inputs" to this layer you created. You're passing the inputs to the dense layer and get x as the output.

Let's add a few more layers to the graph of layers:

```
x = layers.Dense(64, activation = 'relu')(x)
outputs = layers.Dense(10)(x)
```

At this point, you can create a model by specifying its inputs and outputs in the graph of layers:

```
model = keras.Model(input = inputs, output = outputs)
```

This is the basic rule for functional API. Let's take a look at the implementation of content-based filtering.

The implementation of content-based filtering:

```
# build the neural network
user = tf.keras.models.Sequential([
    tf.keras.layers.Dense(256, activation = "relu"),
    tf.keras.layers.Dense(128, activation = "relu"),
    tf.keras.layers.Dense(32)
])
u_in = tf.keras.layers.Input(shape = (num_user_features))
# u_in ----> user ----> u_out
u_out = user(u_in)
# L2 normalization(devides the vector by its length)
u_out = tf.linalg.l2_normalize(u_out, axis = 1) # along
asix 1

# build two separate sequentail network although they
have the same structure.
item = tf.keras.models.Sequential([
    tf.keras.layers.Dense(256, activation = "relu"),
    tf.keras.layers.Dense(128, activation = "relu"),
    tf.keras.layers.Dense(32)
])
i_in = tf.keras.layers.Input(shape = (num_item_features))
# i_in ----> user ----> i_out
i_out = item(i_in)
i_out = tf.linalg.l2_normalize(i_out, axis = 1)

# add a dot layer. (along axis 1)
output = tf.keras.layers.Dot(axes = 1)([u_out, i_out])
model = tf.keras.Model([u_in, i_in], output) # sepecify
the input layer and output layer.
model.compile(
    optimizer = tf.optimizers.Adam(learning_rate = 0.01),
    loss = tf.keras.losses.MeanSquaredError()
)
```

Reinforcement learning

Concepts

Reinforcement learning is a type of machine learning where an agent learns to make decisions by interacting with the environment. Key components of reinforcement learning include:

- Agent: The entity that make decisions and take actions in the environment.
- State: A representation of the current environment.
- Action: The possible moves that agent can make.
- Reward: A numerical value that the environment provide to the agent as feedback for the action.

Bellman equation

Based on dynamic programming, we can define $Q(s, a)$ as the maximum profits. It means that we start in s, take action a and behave optimally after that. The transition equation is called Bellman equation:

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

Deep Q-Network

Deep Q-Network is a type of neural network used in reinforcement learning to approximate the Q-function. Normal Q-function can not be used in continuous situations, in which case neural network fits right in.

We can train our neural network iteratively by using Bellman equation. The iterative method converges to the optimal action-value function as the **episode** is big enough.

The following three methods are key components of the algorithm.

Experience Replay

When an agent interacts with the environment, the states / actions / rewards are sequential by nature. If the agent tries to learn from these consecutive experience, it can run into problems due to the strong relationship between them. To avoid this, we use a technique called `Experience Replay`.

We can store all our experience `(state, action, next_state, reward, done)` into a data structure. During the training process, we randomly sample a mini-batch of experiences from the data structure and put it into neural network.

Target Network

Assume that $\widehat{Q}(s,a)$ is converged to the optimal action-value function. The $\widehat{Q}(s,a)$ is called target network. Subsequently, we can use target network to update Q-network $Q(s,a)$:

$$\overbrace{\underbrace{R(s)+\gamma \max_{a'} \widehat{Q}(s',a')}_{\text{y target}} - Q(s,a)}^{\text{Error}}$$

Gradient descent can be applied to the Q-network. Next, we fine tune the target network with Q-network:

$$\widehat{w} \leftarrow \lambda w - (1-\lambda)\widehat{w}$$

where $\lambda << 1$. By using the soft update above, we are ensuring the target network changes slowly, which greatly improves the stability of the training process.

epsilon-greedy

This policy encourages our algorithm to explore for more possibilities. Specifically, we set a threshold ϵ as the possibility to take random steps. Otherwise we act according to the Q-function.

Initially, we set $\epsilon = 1$, which means we take random steps because we don't know anything. As the training process pushes forward, we decrease the ϵ slowly because we believe the Q-function can make the reasonable choice.

Implementations

Take a look at the `Luner Lander` directory.