```
plt.scatter(customers[:,0], customers[:,1])
plt.xlabel("income")
plt.ylabel("debt")
plt.show()
```

[Hint-2] You may want to consider a baseline in specifying the term "risky".

[Hint-3] This problem is only half-way data analytics, to be honest. The issue is still on the concept of "risky". Can you discuss with this respect?

Linear model -- definition and implementation

Now you may have motivated the design of a linear model by yourself! Just assign a weight to each attributes so they together produce a sigle number expressing some view towards a target of analytics.

With such motivation, let's study the linear model's implementation and variants.

In [ ]:

```python
# Here is our linear model. It will work for data with 2 attributes.
linear_model_w0 = 0.6  # Note index starts from 0 as a widely accepted
linear_model_w1 = 0.8
linear_model_b  = 1.5
```

In [ ]:

```python
x = [0.7, -0.2]  # A data sample, check the "List" section in notebook
y = linear_model_w0 * x[0] + linear_model_w1 * x[1] + linear_model_b

print("Linear model: x", x, "->", y)

# if you get a
#    NameError: name 'linear_model_w0' is not defined
# check if the above cell, defining those stuff, has been executed
```

1.2 Model Family and Learning

Well, we can define our linear models. In practice, we actually operates with another concept -- *model family*. Each model is a very particular theory about the relationship between the attributes of samples and the corresponding targets. Such a theory is called a *hypothesis*. However, any particular hypothesis may of little use in practice: say you have a linear model with $w_1 = 0.1573, w_2 = -1.2856$ and $b = -0.0021$ it is hard to imagine the model is optimal for any practical 2D data analytic task. So in the business of data analytics, we adopt the *learning* approach -- where instead of focusing on an individual data model, we define a *model family*, which consists of many, often infinitely many, hypothesis.

HYPOTHESIS SPACE:$\mathcal{H}$

In [ ]:
```
# Let's make a template from which we can realise a family of linear m
class LinearModel2D:
    def __init__(self, w0, w1, b):
        """
        :param w0, w1: init values of the weights for two attributes.
        :param b: init value of the bias
        """
        self.w0 = w0
        self.w1 = w1
        self.b  = b

    def evaluate(self, x):
        """
        :param x: a 2D data sample.
        NB: As long as x allows you to extract its 2 attributes using
        x[0] and x[1] (see code below), it doesn't matter what is its
        specific type.

        ** Function arguments are matched by functionality, not
        type definition ** I feel it makes life easier to implement
        data analytic algorithms. The feature is called "duck typing"
        (Google the term)
        """
        return self.w0 * x[0] + self.w1 * x[1] + self.b

    def __str__(self):
        """
        For pretty print: when you print(a), the actual process is pri
        """
        return "Linear Model 2D: w0={}, w1={}, b={}".format(self.w0, s
```

In [ ]:
```
# let's make one linear model
model_1 = LinearModel2D(0.6, 0.8, 0.5)
model_2 = LinearModel2D(0.6, 1.0, 0.5)

print("Model 1:")
print(x, ":", model_1.evaluate(x))  # compare with the result above
print("Model 2:")
print(x, ":", model_2.evaluate(x))  # compare with the result above
# another data sample
x1 = [0.7, 0]
print(x1, ":", model_1.evaluate(x1))  # compare with the result above
```

Q2 Try another linear model by yourself.

Targets

Now let re-think the Hint-3 in **Q1**: the target of analytics should be given for sample examples. This piece of information, or the lack of it, defines many different types of machine learning. If you think about the situation carefully, there is no essential distinction between "targets" or "attributes" from the point of view of an object. The distinction is made on the model learner's side. Targets are a special set of attributes, which are accessible during the stage we adjust our model. Once the model has been fixed and deployed, the "target" variables are no longer available to the model, for its namesake, they becomes the "target" of the model prediction.

From the various ways the targets are given (or missing) arise supervised (we will see shortly), unsupervised, reinforcement, transfer, ..., learning schemes.

Given the hypotheses in some $\mathcal{H}$, and in the light of data samples drawn from $\mathcal{D}$, we perform some process to pick up the one that mostly fits. The process is called *learning*.

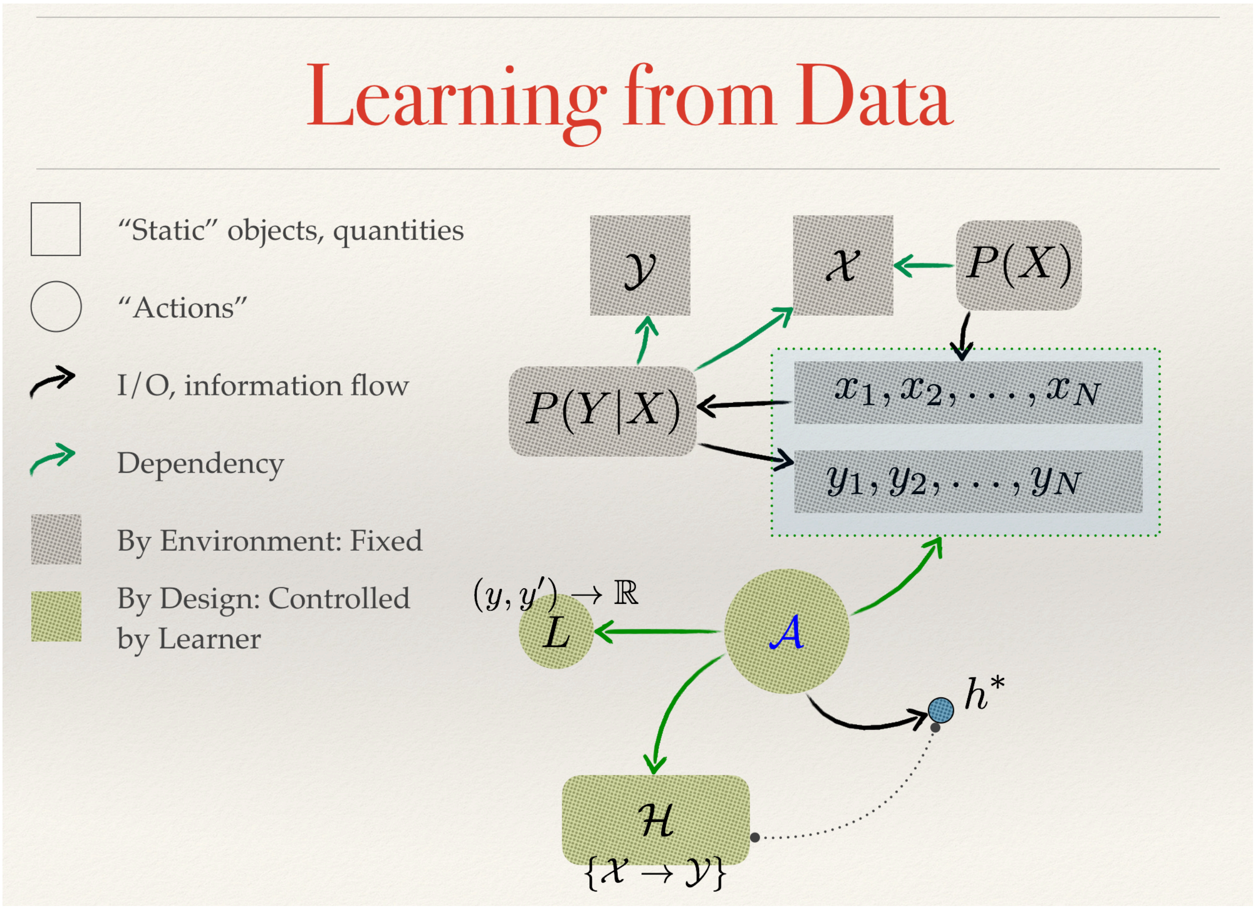
LEARNING ALGORITHM:$\mathcal{A}$

The purpose of learning is of course to make the model's prediction on the targets, given the observable attributes, better aligned with that of the true data. Technically, we need a single criterion, you can consider it as the "KPI" of the model. We optimise over the model parameters (i.e. picking up a specific model/hypothesis from $\mathcal{H}$) to have the best KPI-measurement on the *training set* of data. By convention, the criterion is often formulated as the discrepancy between the desired target and the model prediction, which is to be *minimised*.

LOSS:$\mathcal{L}$.

By now we have encountered all elements in learning-based machine intelligence $(\mathcal{D}, \mathcal{H}, \mathcal{L}, \mathcal{A})$ -- if you accept the view of intelligence as "capable of summarising from past experience to shape behaviour for future reward".

The learning framework is shown in the picture below:



Q3 Can you identify a key assemption in the entire formulation of the problem?

[Hint] The learning is on *training set*, while the ultimate goal is the model performance on $\mathcal{D}$, which is unknown. So we have to hope the training set is representative enough for $\mathcal{D}$, at least, in terms of the aspects that are concerned by $\mathcal{L}$.

In [ ]:
```
# A toy example.
x_set = [[-0.1, 0.5],
         [0.2, 0.1],
         [-0.3, -0.3],
         [-0.4, 0.4],
         [0.1, 0.2],
         [-0.5, 0.5]]
targets = [+1, +1, -1, -1, +1, -1]
```

In [ ]:
```
print(model_1)
for x in x_set:
    y = model_1.evaluate(x)
    print(x, ":", y)
```

Q4

- Apply another model to the set, see result
- Adjust model parameters (so you will have multiple models), check those model's output on one data point.

Numpy implementation

Refer to the application of numpy in our PyGym. How can you implement the linear model using numpy? Let us represent sample as $[x_0, x_1, \ldots, x_{d-1}]$, and stack multiple samples, so that each one being a *row* in a *data matrix*:

$$\mathbf{X} = \begin{bmatrix} x_{0,0} & x_{0,1} & \ldots & x_{0,d-1} \\ x_{1,0} & x_{1,1} & \ldots & x_{1,d-1} \\ & & \ldots & \\ x_{n-1,0} & x_{n-1,1} & \ldots & x_{n-1,d-1} \end{bmatrix}$$

NB, we use 2 subscriptors for x now, the first one as sample ID, the second one for attributes. I.e. the *second* subscript is corresponding to the only subscript we used above for a single sample.

NB2, in a 2D case, the X-mat is simplified as

$$\mathbf{X} = \begin{bmatrix} x_{0,0} & x_{0,1} \\ x_{1,0} & x_{1,1} \\ \ldots & \\ x_{n-1,0} & x_{n-1,1} \end{bmatrix}$$

Now, let us have our $\boldsymbol{w} = [w_0, w_1]$, forget about the bias b for the moment, the linear model evaluation on each of the samples, i.e. each row of the matrix $\mathbf{X}$, is

$$\begin{aligned} y_0 &= x_{0,0}w_0 + x_{0,1}w_1 + \cdots + x_{0,d-1}w_{d-1} \\ y_1 &= x_{1,0}w_0 + x_{1,1}w_1 + \cdots + x_{1,d-1}w_{1,d-1} \\ &\ldots \\ y_n - 1 &= x_{n-1,0}w_0 + x_{n-1,1}w_1 + \cdots + x_{n-1,d-1}w_{d-1} \end{aligned}$$

NB2a: replace d with 2 and re-write the equation yourself, see how it works for a simple 2D case. (You can simply double-click here to edit this cell, and copy-and-paste the above equation below, deleting what is not needed when $d = 2$, then pretty-display this cell again by executing it).

The above computation is exactly a matrix-by-vector multiplication $\boldsymbol{y} = \mathbf{X} \cdot \boldsymbol{w}$, and is straightforwardly corresponding to numpy array operation [dot (http://numpy-dot-documnet)](http://numpy-dot-documnet).

Q5 *

Perform matrix computation and link to perceptron training Please try to compute:

```
[[-0.1,  0.5, 1.0],          [0.6,
 [ 0.2,  0.1, 1.0],    *      0.8,
 [-0.3, -0.3, 1.0],           0.5]
 [-0.4,  0.4, 1.0]]
```

Hint: `[ 0.84   0.7    0.08   0.58]`

In [ ]:
```python
import numpy as np
class LinearModel2D:
    def __init__(self, w0, w1, b):
        """
        w-parameter is now stored as a 1-D array.
        """
        self.w = np.asarray([w0, w1])
        self.b = b

    def evaluate(self, x):
        """
        :param x: one or more 2D data sample.
        """
        x = np.atleast_2d(x)  # if you don't do this, you may have
        # get error when x is a single-sample dataset.

        # Q: how to implement y[i] = w[0]*x[i, 0] + w[1]*x[i, 1] + b
        #    using Python matrix operations?
        y = np.dot(x, self.w) # {tutor}
        y += self.b           # {tutor} and explain the concept of "br
        return y

    def __str__(self):
        """
        For pretty print
        """
        return "Linear Model 2D: w0={}, w1={}, b={}".format(
            self.w[0], self.w[1], self.b)
```

NB

The mathematical convention is to consider a vector as a *column* vector. E.g. if w is a 2D vector, we consider w as $\begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$, rather than $[w_1, w_2]$.

In computer implementation, however, a *vector* is a complex object has only one dimension, i.e. you traverse the elements in such an object, e.g. `w` , using only *one* index, `w[0]` , `w[1]` . So technically, this object doesn't have orientation: it depends on the particular package to interpret it as a COLUMN/ROW. If we want to force an interpretation, we may add a *singular dimension* to the object, a dimension of length 1, thus not adding more contents, but specifying the structure.

E.g. in numpy, we can do `w = w[numpy.newaxis, :]` produces a 1-row matrix, i.e. a row vector, and `w = w[:, numpy.newaxis]` results in a column vector (`newaxis` is a special mark in numpy, replace `numpy` with the name given at importing, usually we use `np`).

In [ ]:
```
# Now let us implement our old friend as an instance of the above defi
lm = LinearModel2D(0.6, 0.8, 0.0)
print(lm)

# Apply model
X = np.asarray(x_set)

A = lm.evaluate(X)
preds = np.sign(A)
for x_, a_, p_, y_ in zip(X, A, preds, targets):  # cf "zipping" in Py
    print ("x {}: lm(x) {}: pred {}: ground-truth {}".format(
        x_, a_, p_, y_))
```

2 Learning: Adapting a linear model to data

Geometrical interpretation of linear model *

It is straightforward to see that with p data attributes, a linear model represents a slope surface in $(p+1)$-dimensional space.

In [ ]:
```
def get_y(x, w, b):
    """
    Solve for y, given x, and line equation
    """
    w1 = w[1] if abs(w[1]) > 1e-9 else 1e-9
    return -(b + w[0]*x) / w1

x0 = -1.0
y0 = get_y(x0, lm.w, lm.b)
# Q6: what do we get if we evaluate the model at (x0, y0)?
#     verify your solution
x1 = 1.0
y1 = get_y(x1, lm.w, lm.b)

plt.plot([x0, x1], [y0, y1]) # link two points (x0, y0) and (x1, y1)
# Q7: what is the meaning of this line?
#     Enable the following block of code and execute again,
#     verify your theory (about the meaning of the line)

ENABLE_DRAW_SAMPLES = True
if ENABLE_DRAW_SAMPLES:
    A = lm.evaluate(X)
    preds = np.sign(A)
    plt.scatter(X[:, 0], X[:, 1], c=targets, s=128, cmap='summer')
    plt.scatter(X[:, 0], X[:, 1], c=preds, s=36, cmap='summer')
    plt.grid('on')
    plt.xlim([-1, +1])
    plt.ylim([-1, +1])
    plt.title("Green: -1; Yellow: +1")
```

Q8

- What does it mean when the colour is consistent?
- What (inner or outer) colour reflects the model prediction?
- What colour reflects the desired value?

Q9

Adjust model and check how it changes the line drawn above (need to re-execute the cell above)

Hint: locate a wrongly classified sample using the grid provided. Try to adjust lm.w and go back the above cell to see effect. E.g. `lm.w += -np.asarray([-0.5, 0.5])`

Perceptron training algorithm

The combination of a linear model family and weight-adjust program (to change prediction on samples where it made mistakes) is called a *perceptron*. The full program is lised below. The central idea is consider the weight $\boldsymbol{w}$ as a direction in the data space of $\mathcal{X}$. (Forget the baseline for now.) Ideally, the inner product between the weight and some data point $\langle \boldsymbol{w}, \boldsymbol{x}_i \rangle$ should be align with the corresponding target value y_i. Be specific,

- for $y_i = -1$: $\langle \boldsymbol{w}, \boldsymbol{x}_i \rangle < 0$
- for $y_i = +1$: $\langle \boldsymbol{w}, \boldsymbol{x}_i \rangle > 0$ what if the current $\boldsymbol{w}$ doesn't give the desired output on some $\boldsymbol{x}_i$?

Q10 * Can you propose a scheme of adjusting $\boldsymbol{w}$? *HINT*: the self-inner product $\langle \boldsymbol{a}, \boldsymbol{a} \rangle$ is always non-negative.

Here is a sketch of the algorithm

```
while there is mis-classified data, say x
    if x: +1, w(t+1) <- w(t) + x
    if x: -1, w(t+1) <- w(t) - x
```

Sketch of proof that the algorithm finds the solution when there is one

1. Consider such an error free $\boldsymbol{w}*$
2. Consider the minimum “safe margin” among all $\mathcal{X}$ by $\boldsymbol{w}^*$, must be some $\rho > 0$.
3. The alignment of $\boldsymbol{w}_t$ with $\boldsymbol{w}^*$ would be improved more than ρ in each step, i.e.
$$\langle \boldsymbol{w}_t, \boldsymbol{w}^* \rangle - \langle \boldsymbol{w}_{t-1}, \boldsymbol{w}^* \rangle > \rho$$
4. The increase of the length of $\boldsymbol{w}_t$ is limited by the data point of largest length.
5. Combine 3. and 4., one can have the *direction* of $\boldsymbol{w}_t$, $\frac{\boldsymbol{w}_t}{\|\boldsymbol{w}_t\|_2}$, aligns with $\boldsymbol{w}^*$ with t increasing
6. Therefore $\boldsymbol{w}_t$ will produce desired symbols on all data points with large t.

A full proof can be found online (https://matthewhr.wordpress.com/2014/05/30/perceptron-convergence-theorem/).

Q11 * Consider what if there is no such perfect data separator $\boldsymbol{w}^*$?

```
In [ ]: def train_perceptron(lm, X, y):
            """
            :param lm: a linear model
            :param X: 2D sample set
            :param y: ground-truth labels
            """
            more = True
            while more:
                more = False
                pred = np.sign(lm.evaluate(X))
                no_fit = np.nonzero(pred!=y)[0] # see PyGym np.nonzeront
                if len(no_fit)>0:
                    x_ = X[no_fit[0]]
                    # Q: here we take the first mis-classified sample.
                    #    how to take a random one?
                    y_ = y[no_fit[0]]
                    lm.w += x_*y_
                    lm.b += y_
                    more = True
                    # NB: lm changed here, you don't need to explicitly
                    #     return and re-assign lm. See PyGym:[Variable scope a
                    #     mutable arguments]
```

```
In [ ]: rng = np.random.RandomState(42)  # random generator
        w = rng.rand(2) * 2.0 - 1.0
        b = rng.rand() * 2.0 - 1.0
        lm = LinearModel2D(w[0], w[1], b)

        # Q: Let us train the model lm.
        train_perceptron(lm, X, targets) # {tutor}
        A = lm.evaluate(X)
        preds = np.sign(A)
        for x_, a_, p_, y_ in zip(X, A, preds, targets):
            print (("X {}, a {}, predicted as {}, ground-truth {}".format(
                x_, a_, p_, y_)))
```

Perceptron is the beginning and building block

Linear models are surprisingly ubiquitous -- perceptrons can be "stacked" together:

1. take a number of perceptrons, each takes the input and produce own output for any data x
2. collect the output of all the perceptrons, and feed into another layer of perceptrons
3. if top layer reached, done, goto 1. otherwise. Such a structure is called multi-layer perceptron (MLP). It has another name: deep neural networks.

Visualising the training process *

In [ ]:
```
import time
def train_perceptron_with_drawing(lm, X, y):
    """
    Draw each training step to have a pretty understanding.
    Try to learn visualisation functions in this function.
    """

    def get_y(x, w, b):
        w1 = w[1] if abs(w[1]) > 1e-9 else 1e-9
        return -(b + w[0]*x) / w1


    def draw_model_and_samples(X, targets, lm, rounds):
        x0 = -np.abs(X[:, 0]).max()
        y0 = get_y(x0, lm.w, lm.b)
        x1 = -x0
        y1 = get_y(x1, lm.w, lm.b)
        A = lm.evaluate(X)
        preds = np.sign(A)
        plt.clf()
        plt.plot([x0, x1], [y0, y1])
        plt.scatter(X[:, 0], X[:, 1], c=targets, s=128, cmap='summer')
        plt.scatter(X[:, 0], X[:, 1], c=preds, s=36, cmap='summer')
        plt.grid('on')
        plt.xlim([x0*1.05, x1*1.05])
        plt.ylim([-np.abs(X[:, 1]).max()*1.05, np.abs(X[:, 1]).max()*1
        plt.title("Round {}: Errors {}: Green: -1; Yellow: +1".format(
                rounds, np.count_nonzero(preds!=targets)))


    more = True
    rounds = 0
    draw_model_and_samples(X, y, lm, rounds)
    plt.show()
    while more:
        time.sleep(0.5)
        more = False
        pred = np.sign(lm.evaluate(X))
        no_fit = np.nonzero(pred!=y)[0] # see PyGym np.nonzeront
        if len(no_fit)>0:
            x_ = X[no_fit[0]]
            y_ = y[no_fit[0]]
            lm.w += x_*y_
            lm.b += y_
            more = True
            rounds += 1
        draw_model_and_samples(X, y, lm, rounds)
        plt.show()
```

In [ ]:
```
rng = np.random.RandomState(42)  # random generator
w = rng.rand(2) * 2.0 - 1.0
b = rng.rand() * 2.0 - 1.0
lm = LinearModel2D(w[0], w[1], b)
train_perceptron_with_drawing(lm, X, targets)
```