lab4_solution

November 4, 2019

1 COMP3222/6246 Machine Learning Technologies (2019/20)

1.1 Lab 4 - Introduction to Tensorflow

This lab is an introduction to the Tensorflow library, a powerful tool to run machine learning algorithms in Python. The Tensorflow library is the backbone of the exercises you will find in lab 5 and lab 6. Its advantages include flexibility, parallel execution, and being a general framework for computation. On top of that, it is a good entry to put in your CV! ## 1. Installation First of all, we need to import the library in Python. Some Python distributions have it included already, if yours does not, you can sidestep the issue and use Google's Collaboratory environment. Still, it can be a good exercise to try and install it on your local machine. In Unix system's you can simply install it by:

```
pip3 install tensorflow
```

Note: We will be using Tensorflow 1 for the time being since Tensorflow 2 has just been released recently and we haven't tested it yet. However you are encouraged to try it. The guide and the tutorials of Tensorflow 2 can be accessed via https://www.tensorflow.org/guide/ and https://www.tensorflow.org/tutorials/ respectively.

After the installation, run this short test to make sure everything is working:

[1]:
```
# These two lines are required to use Tensorflow 1
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()

x = tf.Variable(3, name="x")
y = tf.Variable(2, name="y")
z = tf.Variable(1, name="z")
g = x*y*z+x*x+z

session = tf.Session()
session.run(x.initializer)
session.run(y.initializer)
session.run(z.initializer)
result = session.run(g)
session.close()

print(result)
```

```
WARNING:tensorflow:From C:\Local\anaconda3\envs\MLTech\lib\site-
packages\tensorflow_core\python\compat\v2_compat.py:65:
disable_resource_variables (from tensorflow.python.ops.variable_scope) is
deprecated and will be removed in a future version.
Instructions for updating:
non-resource variables are not supported in the long term
16
```

The code above creates a simple function of three variables, and then runs a Tensorflow session to compute the result.

Exercise 1.1. Modify the code above to compute the value of $f(x, y, z) = x^3 + y^2 + yz + 3$ with $x = -2$, $y = 5$ and $z = 1.2$

[2]:
```
# Solution

x = tf.Variable(-2, dtype=tf.float32, name="x")
y = tf.Variable(5.0, name="y")
z = tf.Variable(1.2, name="z")
g = x*x*x+y*y+y*z+3

session = tf.Session()
session.run(x.initializer)
session.run(y.initializer)
session.run(z.initializer)
result = session.run(g)
session.close()

print(result)
```

```
26.0
```

1.2 2. Linear regression

In Tensorflow, we can easily define operations on whole arrays, matrices and multi-dimensional matrices (aka tensors). In this section, we look at a straightforward implementation of the vanilla linear regression algorithm.

Do you remember the boston house price dataset from lab 2? Let's load it again and do some regression!

[3]:
```
import numpy as np
from sklearn.datasets import load_boston

# load the dataset
boston = load_boston()
m, n = boston.data.shape
boston_features = np.c_[np.ones((m,1)), boston.data]
```

```python
# define the pseudo-inverse equation in tensorflow
X = tf.constant(boston_features, dtype=tf.float32, name="X")
y = tf.constant(boston.target.reshape(-1, 1), dtype=tf.float32, name="y")
Xt = tf.transpose(X)
w = tf.matmul(tf.matrix_inverse(tf.matmul(Xt, X)), y)

# run the computation
with tf.Session() as sess:
    weights = w.eval()
```

```
      ␣
↪---------------------------------------------------------------------------

       InvalidArgumentError                      Traceback (most recent call␣
↪last)

       C:
↪\Local\anaconda3\envs\MLTech\lib\site-packages\tensorflow_core\python\framework\ops.
↪py in _create_c_op(graph, node_def, inputs, control_inputs)
      1609   try:
   -> 1610     c_op = c_api.TF_FinishOperation(op_desc)
      1611   except errors.InvalidArgumentError as e:


       InvalidArgumentError: Dimensions must be equal, but are 14 and 506 for␣
↪'MatMul_1' (op: 'MatMul') with input shapes: [14,14], [506,1].


   During handling of the above exception, another exception occurred:


       ValueError                                Traceback (most recent call␣
↪last)

       <ipython-input-3-0039b9f7e442> in <module>
        11 y = tf.constant(boston.target.reshape(-1, 1), dtype=tf.float32,␣
↪name="y")
        12 Xt = tf.transpose(X)
   ---> 13 w = tf.matmul(tf.matrix_inverse(tf.matmul(Xt, X)), y)
        14
        15 # run the computation


       C:
↪\Local\anaconda3\envs\MLTech\lib\site-packages\tensorflow_core\python\util\dispatch.
↪py in wrapper(*args, **kwargs)
```

```
    178     """Call target, and fall back on dispatchers if there is a␣
↪TypeError."""
    179     try:
--> 180       return target(*args, **kwargs)
    181     except (TypeError, ValueError):
    182       # Note: convert_to_eager_tensor currently raises a ValueError,␣
↪not a


    C:
↪\Local\anaconda3\envs\MLTech\lib\site-packages\tensorflow_core\python\ops\math_ops.
↪py in matmul(a, b, transpose_a, transpose_b, adjoint_a, adjoint_b,␣
↪a_is_sparse, b_is_sparse, name)
   2763     else:
   2764       return gen_math_ops.mat_mul(
-> 2765           a, b, transpose_a=transpose_a, transpose_b=transpose_b,␣
↪name=name)
   2766
   2767


    C:
↪\Local\anaconda3\envs\MLTech\lib\site-packages\tensorflow_core\python\ops\gen_math_ops.
↪py in mat_mul(a, b, transpose_a, transpose_b, name)
   6133   _, _, _op = _op_def_lib._apply_op_helper(
   6134         "MatMul", a=a, b=b, transpose_a=transpose_a,␣
↪transpose_b=transpose_b,
-> 6135                   name=name)
   6136   _result = _op.outputs[:]
   6137   _inputs_flat = _op.inputs


    C:
↪\Local\anaconda3\envs\MLTech\lib\site-packages\tensorflow_core\python\framework\op_def_libra
↪py in _apply_op_helper(self, op_type_name, name, **keywords)
    791         op = g.create_op(op_type_name, inputs, dtypes=None,␣
↪name=scope,
    792                          input_types=input_types, attrs=attr_protos,
--> 793                          op_def=op_def)
    794       return output_structure, op_def.is_stateful, op
    795


    C:
↪\Local\anaconda3\envs\MLTech\lib\site-packages\tensorflow_core\python\util\deprecation.
↪py in new_func(*args, **kwargs)
```

```
    505                  'in a future version' if date is None else ('after␣
↪%s' % date),
    506                  instructions)
--> 507        return func(*args, **kwargs)
    508
    509      doc = _add_deprecated_arg_notice_to_docstring(


    C:
↪\Local\anaconda3\envs\MLTech\lib\site-packages\tensorflow_core\python\framework\ops.
↪py in create_op(***failed resolving arguments***)
   3358          raise TypeError("Input #%d is not a tensor: %s" % (idx, a))
   3359      return self._create_op_internal(op_type, inputs, dtypes,␣
↪input_types, name,
-> 3360                                      attrs, op_def, compute_device)
   3361
   3362    def _create_op_internal(


    C:
↪\Local\anaconda3\envs\MLTech\lib\site-packages\tensorflow_core\python\framework\ops.
↪py in _create_op_internal(self, op_type, inputs, dtypes, input_types, name,␣
↪attrs, op_def, compute_device)
   3427            input_types=input_types,
   3428            original_op=self._default_original_op,
-> 3429            op_def=op_def)
   3430        self._create_op_helper(ret, compute_device=compute_device)
   3431      return ret


    C:
↪\Local\anaconda3\envs\MLTech\lib\site-packages\tensorflow_core\python\framework\ops.
↪py in __init__(self, node_def, g, inputs, output_types, control_inputs,␣
↪input_types, original_op, op_def)
   1771            op_def, inputs, node_def.attr)
   1772        self._c_op = _create_c_op(self._graph, node_def,␣
↪grouped_inputs,
-> 1773                                  control_input_ops)
   1774      # pylint: enable=protected-access
   1775


    C:
↪\Local\anaconda3\envs\MLTech\lib\site-packages\tensorflow_core\python\framework\ops.
↪py in _create_c_op(graph, node_def, inputs, control_inputs)
   1611    except errors.InvalidArgumentError as e:
   1612      # Convert to ValueError for backwards compatibility.
```

```
-> 1613     raise ValueError(str(e))
   1614
   1615   return c_op


     ValueError: Dimensions must be equal, but are 14 and 506 for 'MatMul_1'␣
↪(op: 'MatMul') with input shapes: [14,14], [506,1].
```

Exercise 2.1. The pseudo-inverse equation in the code above is wrong. Fix the error.

[4]:
```python
# Solution

X = tf.constant(boston_features, dtype=tf.float32, name="X")
y = tf.constant(boston.target.reshape(-1, 1), dtype=tf.float32, name="y")
Xt = tf.transpose(X)
w = tf.matmul(tf.matmul(tf.matrix_inverse(tf.matmul(Xt, X)), Xt), y)

# run the computation
with tf.Session() as sess:
    weights = w.eval()

print(weights)
```

```
[[ 3.6492584e+01]
 [-1.0801753e-01]
 [ 4.6391938e-02]
 [ 2.0557338e-02]
 [ 2.6870332e+00]
 [-1.7785543e+01]
 [ 3.8087704e+00]
 [ 7.1130082e-04]
 [-1.4757435e+00]
 [ 3.0610541e-01]
 [-1.2327950e-02]
 [-9.5366442e-01]
 [ 9.3092406e-03]
 [-5.2478248e-01]]
```

Exercise 2.2. Modify the code above to compute some estimates over the training set. Print the training RMSE.

[5]:
```python
# Solution

predict = tf.matmul(X, w)
sq_error = tf.squared_difference(predict, y)
rmse = tf.math.sqrt(tf.reduce_mean(sq_error))

# run the computation
```

```
with tf.Session() as sess:
    RMSE = rmse.eval()

print(RMSE)
```

```
4.679192
```

From the examples seen so far, we can deduce that the Tensorflow library is designed around two phases. First, is the **declaration phase**, where we create all the variables and link them into a function. Internally, this generates a computation graph. Second, we create a Tensorflow session and we run the **actual computation**. ## 3. Gradient descent When the number of features and the dataset are large, computing the pseudo-inverse can become computationally expensive. A more efficient approach is gradient descent, which consists in starting from a randomly selected point and slowly creeping toward the solution. Not only this approach is quick, but it generalises well beyond linear methods. In fact, this is the backbone of the many non-linear neural networks and deep learning algorithms that define the current state-of-the-art.

Here is an example of how to implement gradient descent in Tensorflow. In this case, the gradients are computed automagically by automatic differentiation. This is a quite fascinating computational technique that saves us from computing first-order derivatives with pen and paper. Have a look at the Wikipedia entry to know more about this topic.

[6]:
```
n_steps = 1000
learn_rate = 0.001

X = tf.constant(boston_features, dtype=tf.float32, name="X")
y = tf.constant(boston.target.reshape(-1, 1), dtype=tf.float32, name="y")
w = tf.Variable(tf.random_uniform([n+1,1], -1.0, 1.0), name="w")
y_hat = tf.matmul(X, w, name="y_hat")
error = y_hat - y
mse = tf.reduce_mean(tf.square(error), name="mse")

gradients = tf.gradients(mse, [w])[0]
train_step = tf.assign(w, w - learn_rate * gradients)

init = tf.global_variables_initializer()
with tf.Session() as sess:
    sess.run(init)

    for step in range(n_steps):
        if step % 50 == 0:
            print("Step", step, "MSE =", mse.eval())
        sess.run(train_step)

    w_best = w.eval()
```

```
Step 0 MSE = 55828.516
Step 50 MSE = nan
Step 100 MSE = nan
```

```
Step 150 MSE = nan
Step 200 MSE = nan
Step 250 MSE = nan
Step 300 MSE = nan
Step 350 MSE = nan
Step 400 MSE = nan
Step 450 MSE = nan
Step 500 MSE = nan
Step 550 MSE = nan
Step 600 MSE = nan
Step 650 MSE = nan
Step 700 MSE = nan
Step 750 MSE = nan
Step 800 MSE = nan
Step 850 MSE = nan
Step 900 MSE = nan
Step 950 MSE = nan
```

Exercise 3.1. Add comments to the code above. Do you understand the purpose of each line?

[7]:
```
# Partial solution
# (Please have a look on the online API documentation. They are very useful.)

n_steps = 1000 # number of training epochs
learn_rate = 0.001 # learning rate (see how it's used to update weight below)

X = tf.constant(boston_features, dtype=tf.float32, name="X")
y = tf.constant(boston.target.reshape(-1, 1), dtype=tf.float32, name="y")
w = tf.Variable(tf.random_uniform([n+1,1], -1.0, 1.0), name="w") # get a random␣
↪tensor with uniform distribution
y_hat = tf.matmul(X, w, name="y_hat")
error = y_hat - y
mse = tf.reduce_mean(tf.square(error), name="mse")

gradients = tf.gradients(mse, [w])[0] # compute a gradient of mse with respect␣
↪to the weight vector
train_step = tf.assign(w, w - learn_rate * gradients) # compute and assign the␣
↪new weight to w

init = tf.global_variables_initializer()
with tf.Session() as sess:
    sess.run(init)

    for step in range(n_steps):
        if step % 50 == 0:
            print("Step", step, "MSE =", mse.eval())
        sess.run(train_step)
```

```
    w_best = w.eval()
```

```
Step 0 MSE = 29126.158
Step 50 MSE = nan
Step 100 MSE = nan
Step 150 MSE = nan
Step 200 MSE = nan
Step 250 MSE = nan
Step 300 MSE = nan
Step 350 MSE = nan
Step 400 MSE = nan
Step 450 MSE = nan
Step 500 MSE = nan
Step 550 MSE = nan
Step 600 MSE = nan
Step 650 MSE = nan
Step 700 MSE = nan
Step 750 MSE = nan
Step 800 MSE = nan
Step 850 MSE = nan
Step 900 MSE = nan
Step 950 MSE = nan
```

Exercise 3.2. The gradient descent algorithm is really sensitive to the value of the learning rate. Try changing it by a few orders of magnitude and run the algorithm again.

[8]:
```
# Solution
n_steps = 200000
learn_rate = 0.001

X = tf.constant(boston_features, dtype=tf.float32, name="X")
y = tf.constant(boston.target.reshape(-1, 1), dtype=tf.float32, name="y")
w = tf.Variable(tf.random_uniform([n+1,1], -1.0, 1.0), name="w")
y_hat = tf.matmul(X, w, name="y_hat")
error = y_hat - y
mse = tf.reduce_mean(tf.square(error), name="mse")

gradients = tf.gradients(mse, [w])[0]
train_step = tf.assign(w, w - learn_rate * gradients)

init = tf.global_variables_initializer()
with tf.Session() as sess:
    sess.run(init)

    for step in range(n_steps):
        if step % 5000 == 0:
            print("Step", step, "MSE =", mse.eval())
        sess.run(train_step)
```

```
w_best = w.eval()
```

```
Step 0 MSE = 20972.34
Step 5000 MSE = nan
Step 10000 MSE = nan
Step 15000 MSE = nan
Step 20000 MSE = nan
Step 25000 MSE = nan
Step 30000 MSE = nan
Step 35000 MSE = nan
Step 40000 MSE = nan
Step 45000 MSE = nan
Step 50000 MSE = nan
Step 55000 MSE = nan
Step 60000 MSE = nan
Step 65000 MSE = nan
Step 70000 MSE = nan
Step 75000 MSE = nan
Step 80000 MSE = nan
Step 85000 MSE = nan
Step 90000 MSE = nan
Step 95000 MSE = nan
Step 100000 MSE = nan
Step 105000 MSE = nan
Step 110000 MSE = nan
Step 115000 MSE = nan
Step 120000 MSE = nan
Step 125000 MSE = nan
Step 130000 MSE = nan
Step 135000 MSE = nan
Step 140000 MSE = nan
Step 145000 MSE = nan
Step 150000 MSE = nan
Step 155000 MSE = nan
Step 160000 MSE = nan
Step 165000 MSE = nan
Step 170000 MSE = nan
Step 175000 MSE = nan
Step 180000 MSE = nan
Step 185000 MSE = nan
Step 190000 MSE = nan
Step 195000 MSE = nan
```

Exercise 3.3. Perform some feature scaling on the dataset (see lab 2), and run the gradient descent algorithm again. Do you see any difference in the result? What about the number of steps needed to converge to the optimum?

[9]:

```python
# Solution (with StandardScaler)
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
scaler.fit(boston_features)
scaled_boston_features = scaler.transform(boston_features)

n_steps = 200000
learn_rate = 0.001

scaled_X = tf.constant(scaled_boston_features, dtype=tf.float32, name="X")
y = tf.constant(boston.target.reshape(-1, 1), dtype=tf.float32, name="y")
w = tf.Variable(tf.random_uniform([n+1,1], -1.0, 1.0), name="w")
y_hat = tf.matmul(scaled_X, w, name="y_hat")
error = y_hat - y
mse = tf.reduce_mean(tf.square(error), name="mse")

gradients = tf.gradients(mse, [w])[0]
train_step = tf.assign(w, w - learn_rate * gradients)

init = tf.global_variables_initializer()
with tf.Session() as sess:
    sess.run(init)

    for step in range(n_steps):
        if step % 5000 == 0:
            print("Step", step, "MSE =", mse.eval())
        sess.run(train_step)

    w_best = w.eval()
```

```
Step 0 MSE = 591.42926
Step 5000 MSE = 529.9725
Step 10000 MSE = 529.70197
Step 15000 MSE = 529.64404
Step 20000 MSE = 529.62836
Step 25000 MSE = 529.62396
Step 30000 MSE = 529.6227
Step 35000 MSE = 529.62225
Step 40000 MSE = 529.62213
Step 45000 MSE = 529.6222
Step 50000 MSE = 529.6222
Step 55000 MSE = 529.6221
Step 60000 MSE = 529.62225
Step 65000 MSE = 529.6221
Step 70000 MSE = 529.6221
Step 75000 MSE = 529.6221
```

```
Step 80000 MSE = 529.6221
Step 85000 MSE = 529.6221
Step 90000 MSE = 529.6221
Step 95000 MSE = 529.6221
Step 100000 MSE = 529.6221
Step 105000 MSE = 529.6221
Step 110000 MSE = 529.6221
Step 115000 MSE = 529.6221
Step 120000 MSE = 529.6221
Step 125000 MSE = 529.6221
Step 130000 MSE = 529.6221
Step 135000 MSE = 529.6221
Step 140000 MSE = 529.6221
Step 145000 MSE = 529.6221
Step 150000 MSE = 529.6221
Step 155000 MSE = 529.6221
Step 160000 MSE = 529.6221
Step 165000 MSE = 529.6221
Step 170000 MSE = 529.6221
Step 175000 MSE = 529.6221
Step 180000 MSE = 529.6221
Step 185000 MSE = 529.6221
Step 190000 MSE = 529.6221
Step 195000 MSE = 529.6221
```

[10]:
```python
# Solution (with MinMaxScaler)
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
scaler.fit(boston_features)
scaled_boston_features = scaler.transform(boston_features)

n_steps = 200000
learn_rate = 0.001

scaled_X = tf.constant(scaled_boston_features, dtype=tf.float32, name="X")
y = tf.constant(boston.target.reshape(-1, 1), dtype=tf.float32, name="y")
w = tf.Variable(tf.random_uniform([n+1,1], -1.0, 1.0), name="w")
y_hat = tf.matmul(scaled_X, w, name="y_hat")
error = y_hat - y
mse = tf.reduce_mean(tf.square(error), name="mse")

gradients = tf.gradients(mse, [w])[0]
train_step = tf.assign(w, w - learn_rate * gradients)

init = tf.global_variables_initializer()
with tf.Session() as sess:
```

```python
    sess.run(init)

    for step in range(n_steps):
        if step % 5000 == 0:
            print("Step", step, "MSE =", mse.eval())
        sess.run(train_step)

    w_best = w.eval()
```

```
Step 0 MSE = 609.05414
Step 5000 MSE = 48.04026
Step 10000 MSE = 38.78988
Step 15000 MSE = 34.25912
Step 20000 MSE = 31.802464
Step 25000 MSE = 30.381323
Step 30000 MSE = 29.517126
Step 35000 MSE = 28.968733
Step 40000 MSE = 28.60674
Step 45000 MSE = 28.358526
Step 50000 MSE = 28.181986
Step 55000 MSE = 28.05208
Step 60000 MSE = 27.953484
Step 65000 MSE = 27.876526
Step 70000 MSE = 27.81517
Step 75000 MSE = 27.765299
Step 80000 MSE = 27.724104
Step 85000 MSE = 27.68978
Step 90000 MSE = 27.660774
Step 95000 MSE = 27.636242
Step 100000 MSE = 27.61521
Step 105000 MSE = 27.597176
Step 110000 MSE = 27.581713
Step 115000 MSE = 27.568287
Step 120000 MSE = 27.55665
Step 125000 MSE = 27.546562
Step 130000 MSE = 27.537785
Step 135000 MSE = 27.530024
Step 140000 MSE = 27.523392
Step 145000 MSE = 27.517506
Step 150000 MSE = 27.512278
Step 155000 MSE = 27.507776
Step 160000 MSE = 27.503843
Step 165000 MSE = 27.500345
Step 170000 MSE = 27.497213
Step 175000 MSE = 27.494444
Step 180000 MSE = 27.491974
Step 185000 MSE = 27.489845
```

```
Step 190000 MSE = 27.487984
Step 195000 MSE = 27.486319
```

1.3 4. Principal component analysis

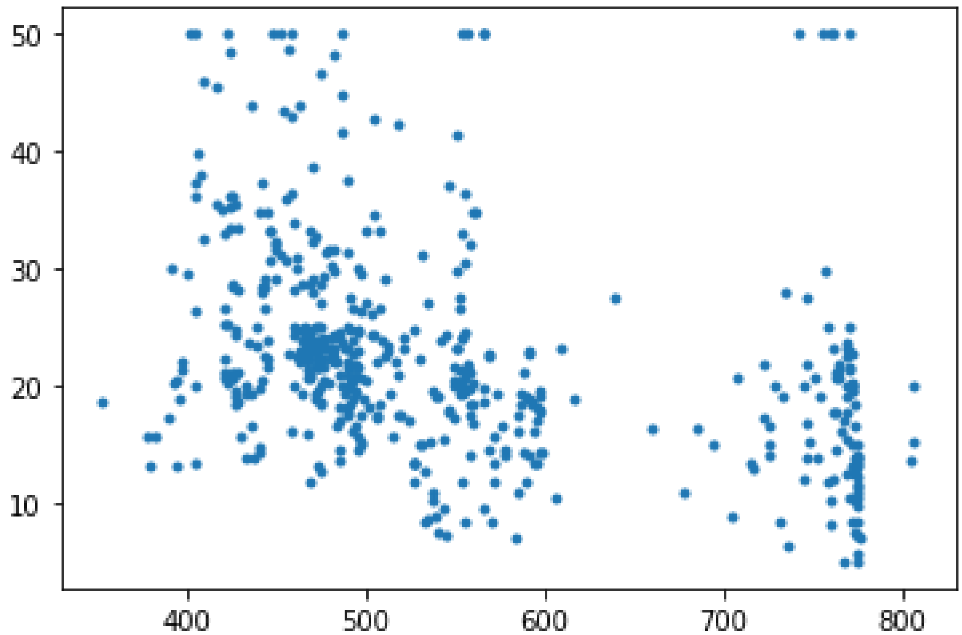
In order to improve our familiarity with Tensorflow, we play with a different topic here. One of the main problem in machine learning is how to visualise multi-dimensional data. In the case of the boston house price dataset, we have 13 input features. Can we plot this 13-dimensional space on a 2-dimensional page somehow?

A possible solution is to use principal component analysis (PCA in short). This is an intriguing linear algebraic method that takes a cloud of multidimensional datapoints and create a new set of axes (aka components). The method extract the components that exhibit the largest variance in the data, thus spreading the datapoints as much as possible.

In the code below, we implement PCA using Tensorflow's built-in singular value decomposition algorithm (SVD):

[12]:
```
import matplotlib.pyplot as plt

s, u, v = tf.svd(X)
P_comp = tf.slice(v, [0, 0], [n + 1, 1])
X_proj = tf.matmul(X, P_comp)

with tf.Session() as sess:
    sess.run(X_proj)
    X_final = X_proj.eval()

plt.figure()
plt.plot(X_final, boston.target, ".")
```

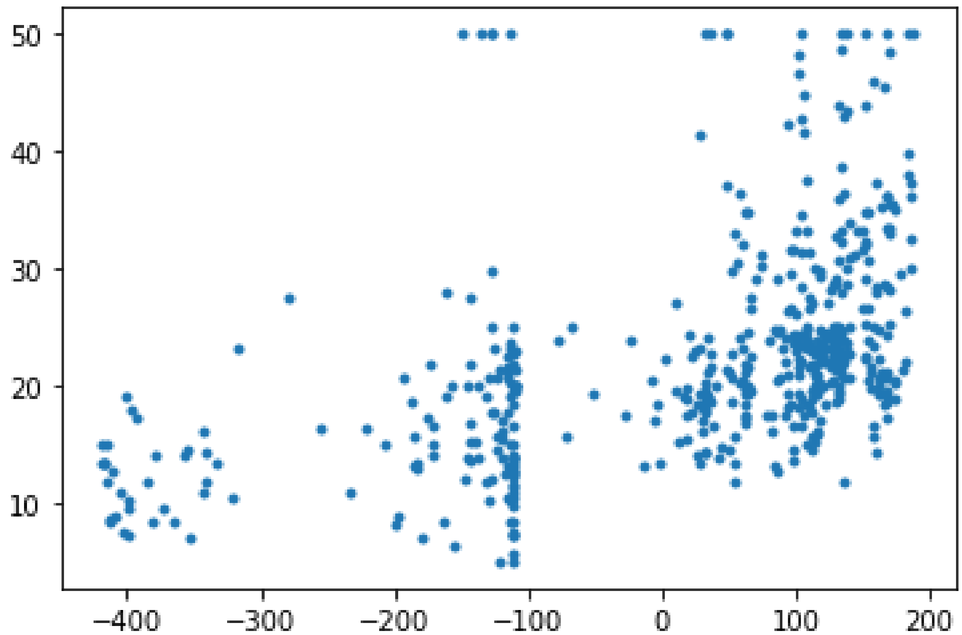
[12]:
```
[<matplotlib.lines.Line2D at 0x29a91191c48>]
```



Exercise 4.1. The code above plots the data along the first principal component. Modify the code to plot along the second.

[13]:
```
# Solution

P_comp2 = tf.slice(v, [0, 1], [n + 1, 1])
X_proj = tf.matmul(X, P_comp2)

with tf.Session() as sess:
    sess.run(X_proj)
    X_final = X_proj.eval()

plt.figure()
plt.plot(X_final, boston.target, ".")
```

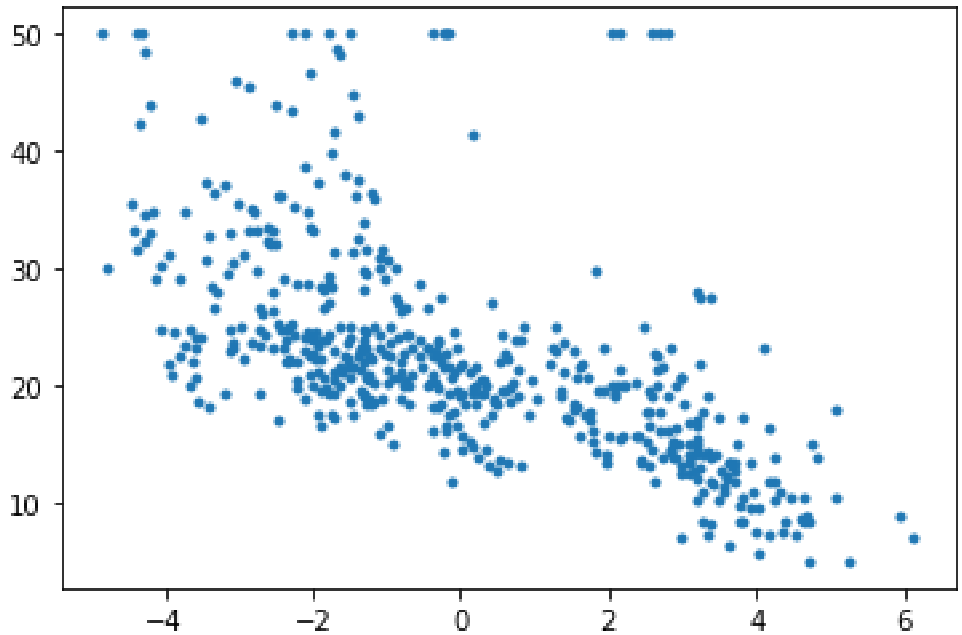
[13]:
```
[<matplotlib.lines.Line2D at 0x29a9121c808>]
```



Exercise 4.2. Does the result change if we perform feature scaling (see lab 2) before running the PCA algorithm?

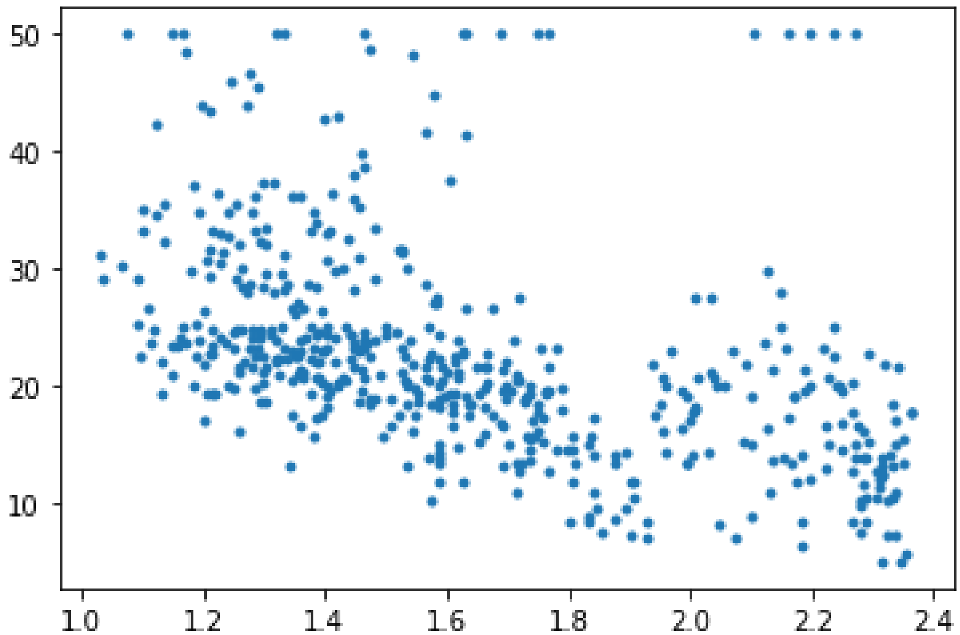
[14]:
```
# Solution (with StandardScaler)
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
scaler.fit(boston_features)
scaled_boston_features = scaler.transform(boston_features)
scaled_X = tf.constant(scaled_boston_features, dtype=tf.float32, name="X")

s, u, v = tf.svd(scaled_X)
P_comp = tf.slice(v, [0, 0], [n + 1, 1])
X_proj = tf.matmul(scaled_X, P_comp)

with tf.Session() as sess:
    sess.run(X_proj)
    X_final = X_proj.eval()

plt.figure()
plt.plot(X_final, boston.target, ".")
```

[14]:
```
[<matplotlib.lines.Line2D at 0x29a912942c8>]
```



[15]:
```
# Solution (with MaxMinScaler)
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
scaler.fit(boston_features)
scaled_boston_features = scaler.transform(boston_features)
scaled_X = tf.constant(scaled_boston_features, dtype=tf.float32, name="X")

s, u, v = tf.svd(scaled_X)
P_comp = tf.slice(v, [0, 0], [n + 1, 1])
X_proj = tf.matmul(scaled_X, P_comp)

with tf.Session() as sess:
    sess.run(X_proj)
    X_final = X_proj.eval()

plt.figure()
plt.plot(X_final, boston.target, ".")
```

[15]:
```
[<matplotlib.lines.Line2D at 0x29a912ffe88>]
```


50
40
30
20
10
1.0
1.2
1.4
1.6
1.8
2.0
2.2
2.4