

# Machine Learning With TensorFlow

X433.7-001 (2 semester units in COMPSCI)

Instructor Alexander I. Iliev, Ph.D.

# Course Content Outline

- **Machine Learning With TensorFlow®**
  - Introduction, Python - pros and cons
  - Python modules, DL packages and scientific blocks
  - Working with the shell, IPython and the editor
  - Installing the environment with core packages
  - Writing “Hello World”
- HW1 (10pts)

- **Tensorflow and TensorBoard basics**
- Linear algebra recap
- Data types in Numpy and Tensorflow
- Basic operations in Tensorflow
- Graph models and structures with Tensorboard

- **TensorFlow operations**
- Overloaded operators
- Using Aliases
- Sessions, graphs, variables, placeholders
- Name scopes

- **Data Mining and Machine Learning concepts**
  - Basic Deep Learning Models, k-Means
  - Linear and Logistic Regression
  - Softmax classification
- HW2 (10pts)

- **Neural Networks**
- Multi-layer Neural Network
- Gradient descent and Backpropagation
- Object recognition with Convolutional Neural Network (CNN)
- Activation Functions

# Data Science

Class 4 ...

*Data Mining and Machine Learning concepts ...*

# Machine Learning

- What is *machine learning*?
- The dictionary defines “to learn” as:
  - To get knowledge of something by study, experience, or being taught
  - To become aware by information or from observation
  - To commit to memory
  - To be informed of or to ascertain
  - To receive instruction

# Machine Learning

- Machine Learning basics:

- What is an *attribute*?

Each **instance that provides the input** to machine learning is **characterized by its values** on a fixed, predefined set of features or attributes

Example:

in the weather, the attribute *temperature* has the values: hot > mild > cold

- Basic attribute data types can be: nominal and numeric

Example:

**string** attributes and **date** attributes are effectively **nominal** and **numeric**

# Machine Learning

- Definitions of “learning” from dictionary:
    - To get knowledge of by study, experience, or being taught
    - To become aware by information or from observation
    - To commit to memory
    - To be informed of, ascertain; to receive instruction
  - Operational definition:
    - What is it ?
- Difficult to measure
- Trivial for computers
- Does a slipper learn?
- Does learning imply intention?

# Machine Learning

- What is *machine learning*?
- An **operational definition** can be formulated in the same way for learning:
  - Things learn when they change their behavior in a way that makes them perform better in the future
- This ties learning to *performance* rather than *knowledge*
- You can **test learning** by observing present behavior and comparing it with past behavior

# Machine Learning

- What is *machine learning*?
- ... We therefore choose the word **training** to denote a mindless kind of learning
- We train animals and even plants, although it would be stretching the word a bit to talk of **training objects** such as slippers, which are **not in any sense alive**
- ... But **learning** is different. Learning implies thinking and purpose.
- **Something that learns has to do so intentionally**
- That is why we wouldn't say that a vine has learned to grow around a trellis in a vineyard—we'd say it has been trained
- **Learning without purpose is merely training**



# Machine Learning

- Supervised Machine Learning:
  - The **majority** of practical machine learning **uses supervised learning**
  - Supervised learning is to have **input variables (x)** and an **output variable (Y)** and we use an algorithm to learn the mapping function from the input to the output, hence finding:
$$Y = f(X)$$
  - The goal is to **approximate the mapping function** so well that when you have new input data (x) that you can **predict the output** variables (Y) for that data
  - It is **supervised learning** because the process of learning from the training dataset can be thought of as a teacher supervising the learning process

# Machine Learning

- Supervised Machine Learning:
  - The two groups of supervised learning are:
    - **Classification**: A classification problem is when the **output variable is a category**, such as “red” or “blue” or “disease” and “no disease”
    - **Regression**: A regression problem is **when the output variable is a real value**, such as “dollars” or “weight”
  - Some popular examples of supervised machine learning algorithms are:
    - **Linear regression** for regression problems.
    - **Random forest** for classification and regression problems.
    - **Support vector machines (SVM)** for classification problems.

# Machine Learning

- Unsupervised Machine Learning:
  - Unsupervised learning is where you only have **input data (X)** and **no corresponding output variables**.
  - The goal for unsupervised learning is to **model the underlying structure** or distribution in the data in order to learn more about the data.
  - These are called unsupervised learning because **unlike supervised learning** above **there is no correct answers and there is no teacher**. Algorithms are left to their own devices to discover and present the interesting structure in the data.

# Machine Learning

- Unsupervised Machine Learning:
  - Unsupervised learning problems can be further grouped into clustering and association problems:
    - **Clustering**: A clustering problem is where you want to discover the inherent **groupings in the data**, such as grouping customers by purchasing behavior.
    - **Association**: An association rule learning problem is where you want to **discover** rules that **describe large portions of your data**, such as people that buy X also tend to buy Y.
  - Some popular examples of unsupervised learning algorithms are:
    - **k-means** for clustering problems.
    - **A priori algorithm** for association rule learning problems.

# Machine Learning

- Semi-Supervised Machine Learning:
  - Problems where we have a large amount of **input data (X)** and only **some of it is labeled (Y)** are called **semi-supervised** learning problems
  - These problems **sit in between** both supervised and unsupervised learning.
  - A good example is a photo archive where only some of the images are labeled, (e.g. dog, cat, person) and the majority are unlabeled.

# Machine Learning

- Semi-Supervised Machine Learning:
  - Many real world machine learning problems fall into this area
  - It can be expensive or time-consuming to label data as it may require access to domain experts
  - Hence unlabeled data is cheap and easy to collect and store

# Machine Learning

- Semi-Supervised Machine Learning:
  - You can use unsupervised learning techniques to discover and learn the structure in the input variables
  - You can also use supervised learning techniques to make best guess predictions for the unlabeled data, feed that data back into the supervised algorithm as training data and use the model to make predictions on new unseen data (*back propagation*)

# The weather problem

- Conditions for playing a certain game

Outlook	Temperature	Humidity	Windy	Play
Sunny	Hot	High	False	No
Sunny	Hot	High	True	No
Overcast	Hot	High	False	Yes
Rainy	Mild	Normal	False	Yes
...	...	...	...	...

A set of learned rules might look like this:

```
If outlook = sunny and humidity = high then play = no
If outlook = rainy and windy = true then play = no
If outlook = overcast then play = yes
If humidity = normal then play = yes
If none of the above then play = yes
```



# The weather problem

- Conditions for playing a certain game

		attributes				
instances		Outlook	Temp	Humidity	Windy	Play
	1	Sunny	Hot	High	False	No
	2	Sunny	Hot	High	True	No
	3	Overcast	Hot	High	False	Yes
	4	Rainy	Mild	High	False	Yes
	5	Rainy	Cool	Normal	False	Yes
	6	Rainy	Cool	Normal	True	No
	7	Overcast	Cool	Normal	True	Yes
	8	Sunny	Mild	High	False	No
	9	Sunny	Cool	Normal	False	Yes
	10	Rainy	Mild	Normal	False	Yes
	11	Sunny	Mild	Normal	True	Yes
	12	Overcast	Mild	High	True	Yes
	13	Overcast	Hot	Normal	False	Yes
	14	Rainy	Mild	High	True	No

# Data Mining

- Simple Examples:
  - The Weather Problem:
    - *instances* in a dataset are characterized by the values of *features*, or *attributes*, that measure different aspects of the instance
    - In our case there are four *attributes*:
      - Outlook,
      - Temperature,
      - Humidity,
      - Windy.
    - The *outcome* is whether to play or not

# Data Mining

- Simple Examples:
  - The Weather Problem:
    - there are 36 possible combinations ( $3 \times 3 \times 2 \times 2 = 36$ )
    - 14 of them are present in the set of input examples
    - These rules are meant to be interpreted in order:
      - If outlook = sunny and humidity = high then play = no
      - If outlook = rainy and windy = true then play = no
      - If outlook = overcast then play = yes
      - If humidity = normal then play = yes
      - If none of the above then play = yes

# The weather problem

- Conditions for playing a certain game

		attributes				
		Outlook	Temp	Humidity	Windy	Play
instances	1	Sunny	Hot	High	False	No
	2	Sunny	Hot	High	True	No
	3	Overcast	Hot	High	False	Yes
	4	Rainy	Mild	High	False	Yes
	5	Rainy	Cool	Normal	False	Yes
	6	Rainy	Cool	Normal	True	No
	7	Overcast	Cool	Normal	True	Yes
	8	Sunny	Mild	High	False	No
	9	Sunny	Cool	Normal	False	Yes
	10	Rainy	Mild	Normal	False	Yes
	11	Sunny	Mild	Normal	True	Yes
	12	Overcast	Mild	High	True	Yes
	13	Overcast	Hot	Normal	False	Yes
	14	Rainy	Mild	High	True	No

Rule:

*if humidity = normal then play = yes*

Correct or incorrect?

# Data Mining

- Simple Examples:
  - The Weather Problem:
    - It is possible to just look for any rules that **strongly associate different attribute values**
    - These are called ***association rules***
    - Some of them are:

```
If temperature = cool                then humidity = normal
If humidity = normal and windy = false then play = yes
If outlook = sunny and play = no      then humidity = high
If windy = false and play = no        then outlook = sunny and
                                      humidity = high
```

# Data Mining

- Simple Examples:
  - The Weather Problem: *Weather Data with Some **Numeric** Attributes*

Outlook	Temperature	Humidity	Windy	Play
Sunny	85	85	false	no
Sunny	80	90	true	no
Overcast	83	86	false	yes
Rainy	70	96	false	yes
Rainy	68	80	false	yes
Rainy	65	70	true	no
Overcast	64	65	true	yes
Sunny	72	95	false	no
Sunny	69	70	false	yes
Rainy	75	80	false	yes
Sunny	75	70	true	yes
Overcast	72	90	true	yes
Overcast	81	75	false	yes
Rainy	71	91	true	no

# Weather data with mixed attributes

- Some attributes have numeric values

Outlook	Temperature	Humidity	Windy	Play
Sunny	85	85	False	No
Sunny	80	90	True	No
Overcast	83	86	False	Yes
Rainy	75	80	False	Yes
...	...	...	...	...

```
If outlook = sunny and humidity > 83 then play = no
```

```
If outlook = rainy and windy = true then play = no
```

```
If outlook = overcast then play = yes
```

```
If humidity < 85 then play = yes
```

```
If none of the above then play = yes
```

# Data Mining

- Simple Examples:
    - The Weather Problem:
      - This scheme must create inequalities involving these new numeric attributes rather than simple equality tests as in the former case.
      - This is called a *numeric-attribute problem*
      - It is a *mixed-attribute* problem, because not all attributes are numeric
      - To compare, the first rule given earlier might take the form:
        - If outlook = sunny and humidity > 83 then play = no
- compare → 

If outlook = sunny and play = no	then humidity = high
----------------------------------	----------------------
- These rules are *classification rules*, they predict the classification



# Classification vs. association rules

- Classification rule:

predicts value of a given attribute (the classification of an example)

```
If outlook = sunny and humidity = high  
    then play = no
```

- Association rule:

predicts value of arbitrary attribute (or combination)

```
If temperature = cool then humidity = normal  
If humidity = normal and windy = false  
    then play = yes  
If outlook = sunny and play = no  
    then humidity = high  
If windy = false and play = no  
    then outlook = sunny and humidity = high
```

# Classification vs. association rules

		attributes				
		Outlook	Temp	Humidity	Windy	Play
instances	1	Sunny	Hot	High	False	No
	2	Sunny	Hot	High	True	No
	3	Overcast			False	Yes
	4	Rainy			False	Yes
	5	Partly cloudy	Normal	Normal	False	Yes
	6	Partly cloudy	Normal	Normal	True	No
	7	Partly cloudy	Cool	Normal	True	Yes
	8	Partly cloudy	Mild	Normal	False	No
	9	Sunny	Cool	Normal	False	Yes
	10	Rainy	Mild	Normal	False	Yes
	11	Sunny	Mild	Normal	True	Yes
	12	Overcast	Mild	High	True	Yes
	13	Overcast	Hot	Normal	False	Yes
	14	Rainy	Mild	High	True	No

**Classification problem:  
predict the "class" value**

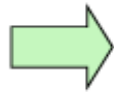
# Weather data with mixed attributes

- Some attributes have numeric values

## Classification

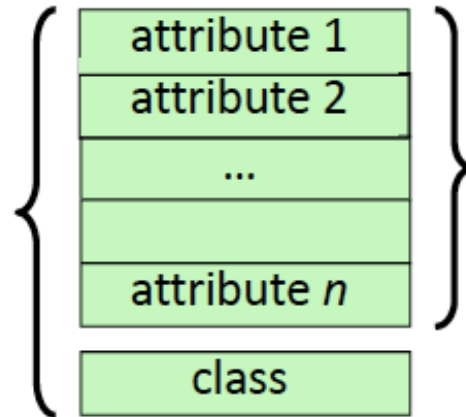
sometimes called “supervised learning”

Dataset: classified examples



“Model” that classifies new examples

classified  
example



instance:  
fixed set of features

discrete (“nominal”)  
continuous (“numeric”)

discrete: “classification” problem  
continuous: “regression” problem

# Classification learning

- Example problems: weather data, contact lenses, irises, labor negotiations
- Classification learning is **supervised**
  - Scheme is provided with **actual outcome**
- Outcome is called the **class** of the example
- Measure success on fresh data for which class labels are known (*test data*)
- In practice success is often measured subjectively

# Association learning

- Can be applied if **no class is specified** and any kind of structure is considered “interesting”
- Difference to classification learning:
  - Can **predict any attribute's value**, not just the class, and more than one attribute's value at a time
  - Hence: far **more association rules** than classification rules
  - Thus: **constraints** are necessary, such as **minimum coverage** and **minimum accuracy**

# Clustering

- Finding groups of items that are similar
- Clustering is **unsupervised**
  - The class of an example is not known
- Success often measured subjectively

	Sepal length	Sepal width	Petal length	Petal width	Type
1	5.1	3.5	1.4	0.2	Iris setosa
2	4.9	3.0	1.4	0.2	Iris setosa
...					
51	7.0	3.2	4.7	1.4	Iris versicolor
52	6.4	3.2	4.5	1.5	Iris versicolor
...					
101	6.3	3.3	6.0	2.5	Iris virginica
102	5.8	2.7	5.1	1.9	Iris virginica
...					

# Numeric prediction

- Variant of classification learning where “class” is **numeric** (also called “**regression**”)
- Learning is **supervised**
  - Scheme is being **provided with target value**
- Measure success on test data

Outlook	Temperature	Humidity	Windy	Play-time
Sunny	Hot	High	False	5
Sunny	Hot	High	True	0
Overcast	Hot	High	False	55
Rainy	Mild	Normal	False	40
...	...	...	...	...

# What's in an example?

- **Instance**: specific type of example
  - **Thing to be classified**, associated, or clustered
  - Individual, independent **example of target concept**
  - Characterized by a **predetermined set of attributes**
- **Input to learning scheme**: set of instances/dataset
  - Represented as a single relation/flat file
- Rather restricted form of input
  - No relationships between objects
- Most common form in practical data mining



# What's in an attribute?

- Each instance is described by a fixed predefined set of features, its **attributes**
- But: number of **attributes may vary** in practice
  - Possible solution: “irrelevant value” flag
- Related problem: existence of an **attribute may depend of value of another** one
- Possible attribute types (“levels of measurement”):
  - *Nominal*, *ordinal*, *interval* and *ratio*

# Machine Learning

- Summary:

attributes = features = predictors

examples = instances = variables

sunny,85,85,FALSE	no	
sunny,80,90,TRUE	no	
overcast,83,86,FALSE	yes	
rainy,70,96,FALSE	yes	
rainy,68,80,FALSE	yes	
rainy,65,70,TRUE	no	
overcast,64,65,TRUE	yes	
sunny,72,95,FALSE	no	
sunny,69,70,FALSE	yes	
rainy,75,80,FALSE	yes	
sunny,75,70,TRUE	yes	
overcast,72,90,TRUE	yes	
overcast,81,75,FALSE	yes	
rainy,71,91,TRUE	no	

class = outcome

# Clustering

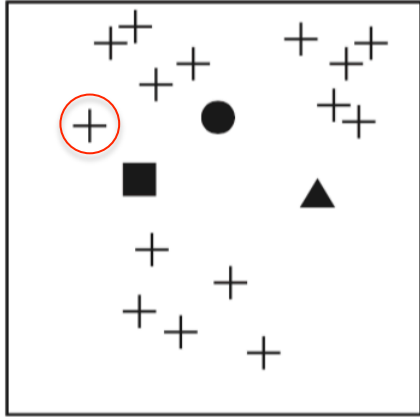
- Clustering techniques **apply when there is no class** to be predicted: they perform **unsupervised learning**
- Aim: **divide instances** into “natural” groups
- Clusters can be:
  - **disjoint vs. overlapping**
  - **deterministic vs. probabilistic**
  - **flat vs. hierarchical**
- We will look at a classic clustering algorithm called ***k-means***
- ***k-means* clusters** are **disjoint, deterministic, and flat**

# The $k$ -means algorithm

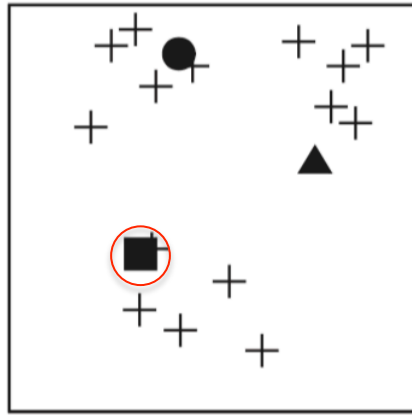
- Step 1: Choose  $k$  random cluster centers
- Step 2: Assign each instance to its closest cluster center based on Euclidean distance
- Step 3: Re-compute cluster centers by computing the average (aka *centroid*) of the instances pertaining to each cluster
- Step 4: If cluster centers have moved, go back to Step 2
- This algorithm minimizes the squared Euclidean distance of the instances from their corresponding cluster centers
  - Determines a solution that achieves a local minimum of the squared Euclidean distance
- Equivalent termination criterion: stop when assignment of instances to cluster centers has not changed

# The *k*-means algorithm: example

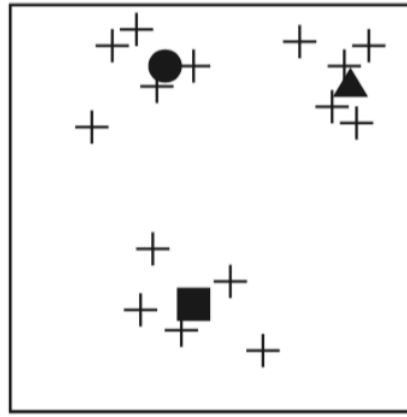
Initial step



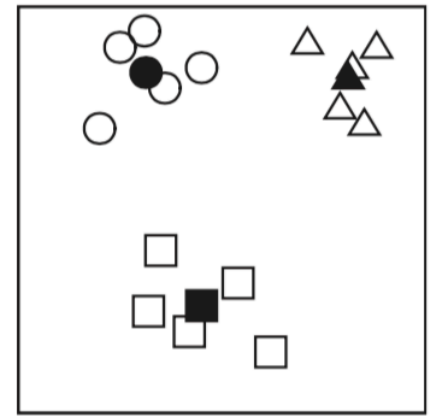
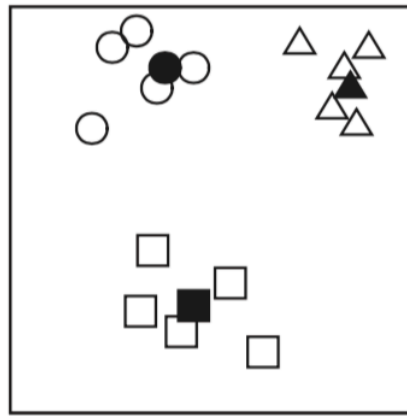
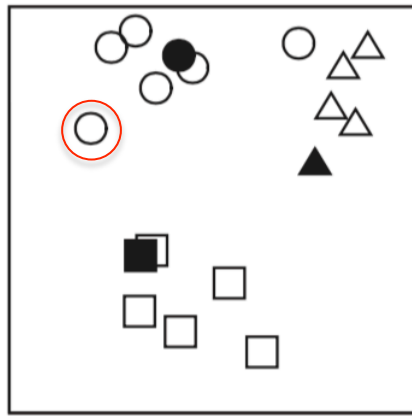
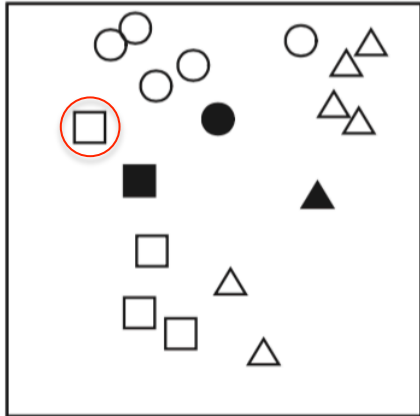
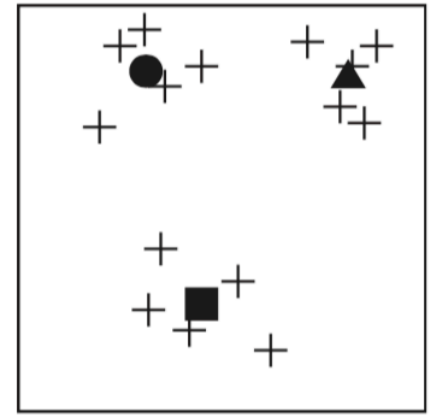
Step 1



Step 2



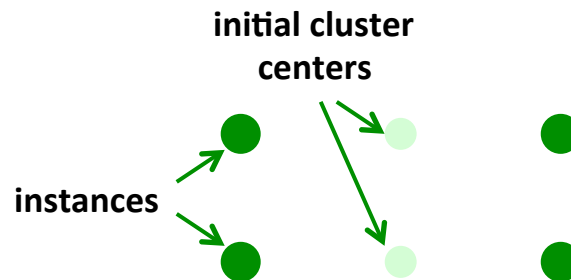
Final step



# Discussion

- Algorithm minimizes squared distance to cluster centers
- Result can vary significantly based on initial choice of seeds
- Can get trapped in local minimum

– Example:



- To increase chance of finding global optimum:  
restart with different random seeds
- Can be applied recursively with  $k = 2$


# Faster distance calculations

- Can we use **kD-trees** or **ball trees** to speed up the process?
- Yes, we can:
  - First, **build the tree data structure**
  - At each node, store the number of instances and the sum of all instances (**summary statistics**)
  - In each iteration of *k*-means, **descend the tree** and find out **which cluster each node belongs to**
  - Can **stop** descending as soon as we find out that a **node belongs entirely to a particular cluster**  
$$\text{Stop} = \text{iterations} * \text{clusters} * \text{instances} * \text{dimensions}$$
  - Use **summary statistics** stored previously at the nodes **to compute new cluster centers**

# K-means with TensorFlow

- Example:

```
1  ## k-means algorithm
2  import matplotlib.pyplot as plt
3  import numpy as np
4  import tensorflow as tf
5
6  # Create aliases:
7  tf.sub = tf.subtract
8
9  points_n = 200
10 clusters_n = 3
11 iteration_n = 100
12
13 # 1. Let's generate random data points with a uniform distribution and assign them
14 #    to a 2D tensor constant. Then, randomly choose initial centroids from the set
15 #    of data points:
16 points = tf.constant(np.random.uniform(0, 10, (points_n, 2)))
17 centroids = tf.Variable(tf.slice(tf.random_shuffle(points), [0, 0], [clusters_n, -1]))
18
19 # 2. Next we want to be able to do element-wise subtraction of points and centroids
20 #    that are 2D tensors. Because the tensors have different shape, let's expand points
21 #    and centroids into 3 dimensions, which allows us to use the broadcasting feature of
22 #    subtraction operation:
23 points_expanded = tf.expand_dims(points, 0)
24 centroids_expanded = tf.expand_dims(centroids, 1)
```



```
(<module>)>>> sess.run(points).shape
(200, 2)

(<module>)>>> sess.run(points_expanded).shape
(1, 200, 2)

(<module>)>>> sess.run(centroids).shape
(3, 2)

(<module>)>>> sess.run(centroids_expanded).shape
(3, 1, 2)
```



# K-means with TensorFlow

- Example:

```

34 # 3. Then, calculate the distances between points and centroids and determine the
35 #    cluster assignments:
36 distances = tf.reduce_sum(tf.square(tf.sub(points_expanded, centroids_expanded)), 2)
37 assignments = tf.argmin(distances, 0)
38
39 # 4. Next, we can compare each cluster with a cluster assignments vector, get points
40 #    assigned to each cluster, and calculate mean values. These mean values are refined
41 #    centroids, so let's update the centroids variable with the new values:
42 means = []
43 for c in range(clusters_n):
44     means.append(tf.reduce_mean( tf.gather(points,
45                                     tf.reshape(tf.where(tf.equal(assignments, c)), [1, -1])), reduction_indices=[1]))
46
47 new_centroids = tf.concat(means, 0)
48
49 update_centroids = tf.assign(centroids, new_centroids)
50 init = tf.global_variables_initializer()

```

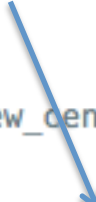
```
(<module>)>>> sess.run(assignments)
array([[1, 2, 2, 0, 2, 1, 2, 1, 0, 2, 0, 2, 2, 0, 2, 1, 1, 1, 1, 2, 2,
        0, 1, 2, 2, 0, 1, 1, 0, 1, 0, 0, 1, 2, 2, 0, 2, 1, 0, 1, 0, 0, 2,
        2, 1, 2, 1, 1, 2, 0, 0, 1, 2, 1, 0, 2, 1, 2, 1, 1, 0, 2, 1, 1,
        1, 1, 1, 2, 2, 1, 2, 2, 1, 1, 2, 1, 0, 1, 2, 1, 0, 1, 0, 1, 2, 2,
        0, 2, 1, 1, 2, 2, 0, 1, 1, 2, 2, 0, 2, 1, 2, 1, 1, 2, 2, 0, 1,
        1, 0, 1, 1, 2, 1, 1, 0, 2, 1, 2, 1, 2, 1, 0, 1, 1, 2, 1, 1,
        1, 0, 2, 0, 0, 1, 2, 0, 0, 1, 2, 1, 1, 1, 2, 0, 1, 1, 0, 1,
        2, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 2, 2, 1, 0, 1, 1, 1, 2, 0,
        1, 2, 1, 1, 2, 1, 2, 2, 2, 1, 1, 0, 1, 1, 0, 2, 2, 1, 1, 0, 1, 0,
        1, 1])
```

```
(<module>)>>> sess.run(tf.equal(assignments, c))
array([False,  False,  False,  True,  False,  False,  False,  False,  True,
        False,  True,  False,  False,  True,  False,  False,  False,  False,
        False,  False,  False,  False,  True,  False,  False,  False,  True,
        False,  False,  True,  False,  True,  True,  False,  False,  False,
        False,  False,  False,  False,  False,  True,  True,  False,  False,
        False,  True,  True,  False,  False,  False,  False,  False,  True,
        False,  False,  False,  False,  False,  False,  False,  False,  False,
        False,  False,  False,  False,  False,  False,  True,  False,  False,
        False,  True,  False,  True,  False,  False,  False,  True,  False,
        False,  False,  False,  False,  True,  False,  False,  False,  False,
        True,  False,  False,  False,  False,  False,  False,  False,  False,
        True,  False,  False,  True,  False,  False,  False,  False,  False])
```

# K-means with TensorFlow

- Example:

```
34 # 3. Then, calculate the distances between points and centroids and determine the
35 #     cluster assignments:
36 distances = tf.reduce_sum(tf.square(tf.sub(points_expanded, centroids_expanded)), 2)
37 assignments = tf.argmin(distances, 0)
38
39 # 4. Next, we can compare each cluster with a cluster assignments vector, get points
40 #     assigned to each cluster, and calculate mean values. These mean values are refined
41 #     centroids, so let's update the centroids variable with the new values:
42 means = []
43 for c in range(clusters_n):
44     means.append(tf.reduce_mean(tf.gather(points,
45                                     tf.reshape(tf.where(tf.equal(assignments, c)), [1, -1])), reduction_indices=[1]))
46
47 new_centroids = tf.concat(means, 0)
48
49 update_centroids = tf.assign(centroids, new_centroids)
50 init = tf.global_variables_initializer()
```



```
(<module>)>>> sess.run(tf.reshape(tf.where(tf.equal(assignments, c)), [1, -1])).shape
(1, 45)
```

```
(<module>)>>> sess.run(tf.reshape(tf.where(tf.equal(assignments, c)), [1, -1]))
array([[ 3,  8, 10, 13, 22, 26, 29, 31, 32, 36, 39, 41, 42,
        50, 51, 55, 56, 62, 78, 82, 84, 88, 94, 99, 108, 111,
        117, 124, 133, 135, 136, 141, 149, 152, 156, 157, 159, 161, 162,
        170, 175, 187, 190, 195, 197]])
```

# K-means with TensorFlow

- Example:

```

34 # 3. Then, calculate the distances between points and centroids and determine the
35 # cluster assignments:
36 distances = ... 2)
37 assignmer (<module>)>>> sess.run(tf.equal(assignments, c))
38 array([[False, False, False, True, False, False, False, False, False, True,
39 # 4. Next
40 # assi
41 # cent
42 means = |
43 for c in
44 means:
45 False, True, True, False, False, False, False, False, False, True,
46 False, False, False, False, False, False, False, True, False, False,
47 new_centri
48 False, False, False, False, True, False, False, False, False, False,
49 update_ce
50 init = ti
    True, False, False, True, False, False, False, False, False, False,

(<module>)>>> sess.run(tf.reshape(tf.where(tf.equal(assignments, c)), [1, -1])).shape
(1, 45)

(<module>)>>> sess.run(tf.reshape(tf.where(tf.equal(assignments, c)), [1, -1]))
array([[ 3,  8, 10, 13, 22, 26, 29, 31, 32, 36, 39, 41, 42,
        50, 51, 55, 56, 62, 78, 82, 84, 88, 94, 99, 108, 111,
        117, 124, 133, 135, 136, 141, 149, 152, 156, 157, 159, 161, 162,
        170, 175, 187, 190, 195, 197]])

```

# K-means with TensorFlow

- Example:

```
34 # 3. Then, calculate the distances between points and centroids and determine the
35 #     cluster assignments:
36 distances = tf.reduce_sum(tf.square(tf.sub(points_expanded, centroids_expanded)), 2)
37 assignments = tf.argmin(distances, 0)
38
39 # 4. Next, we can compare each cluster with a cluster assignments vector, get points
40 #     assigned to each cluster, and calculate mean values. These mean values are refined
41 #     centroids, so let's update the centroids variable with the new values:
42 means = []
43 for c in range(clusters_n):
44     means.append(tf.reduce_mean( tf.gather(points,
45                                     tf.reshape(tf.where(tf.equal(assignments, c)),[1,-1])),reduction_indices=[1]))
46
47 new_centroids = tf.concat(means, 0)
48
49 update_centroids = tf.assign(centroids, new_centroids)
50 init = tf.global_variables_initializer()
```

```
(<module>)>>> sess.run((tf.reduce_mean( tf.gather(points,
        tf.reshape(tf.where(tf.equal(assignments, c)),[1,-1])),reduction_indices=[1]))).shape
(1, 2)


(<module>)>>> sess.run((tf.reduce_mean( tf.gather(points,
        tf.reshape(tf.where(tf.equal(assignments, c)),[1,-1])),reduction_indices=[1]))
array([[1.97671635, 7.45052149]])
```



# K-means with TensorFlow

- Example:

```
34 # 3. Then, calculate the distances between points and centroids and determine the
35 #     cluster assignments:
36 distances = tf.reduce_sum(tf.square(tf.sub(points_expanded, centroids_expanded)), 2)
37 assignments = tf.argmin(distances, 0)
38
39 # 4. Next, we can compare each cluster with a cluster assignments vector, get points
40 #     assigned to each cluster, and calculate mean values. These mean values are refined
41 #     centroids, so let's update the centroids variable with the new values:
42 means = []
43 for c in range(clusters_n):
44     means.append(tf.reduce_mean( tf.gather(points,
45                                     tf.reshape(tf.where(tf.equal(assignments, c)), [1, -1])), reduction_indices=[1]))
46
47 new_centroids = tf.concat(means, 0)
48
49 update_centroids = tf.assign(centroids, new_centroids)
50 init = tf.global_variables_initializer()
```



```
(<module>)>>> sess.run(distances).shape
(3, 200)

(<module>)>>> sess.run(assignments).shape
(200,)

(<module>)>>> sess.run(new_centroids).shape
(3, 2)

(<module>)>>> sess.run(update_centroids).shape
(3, 2)
```

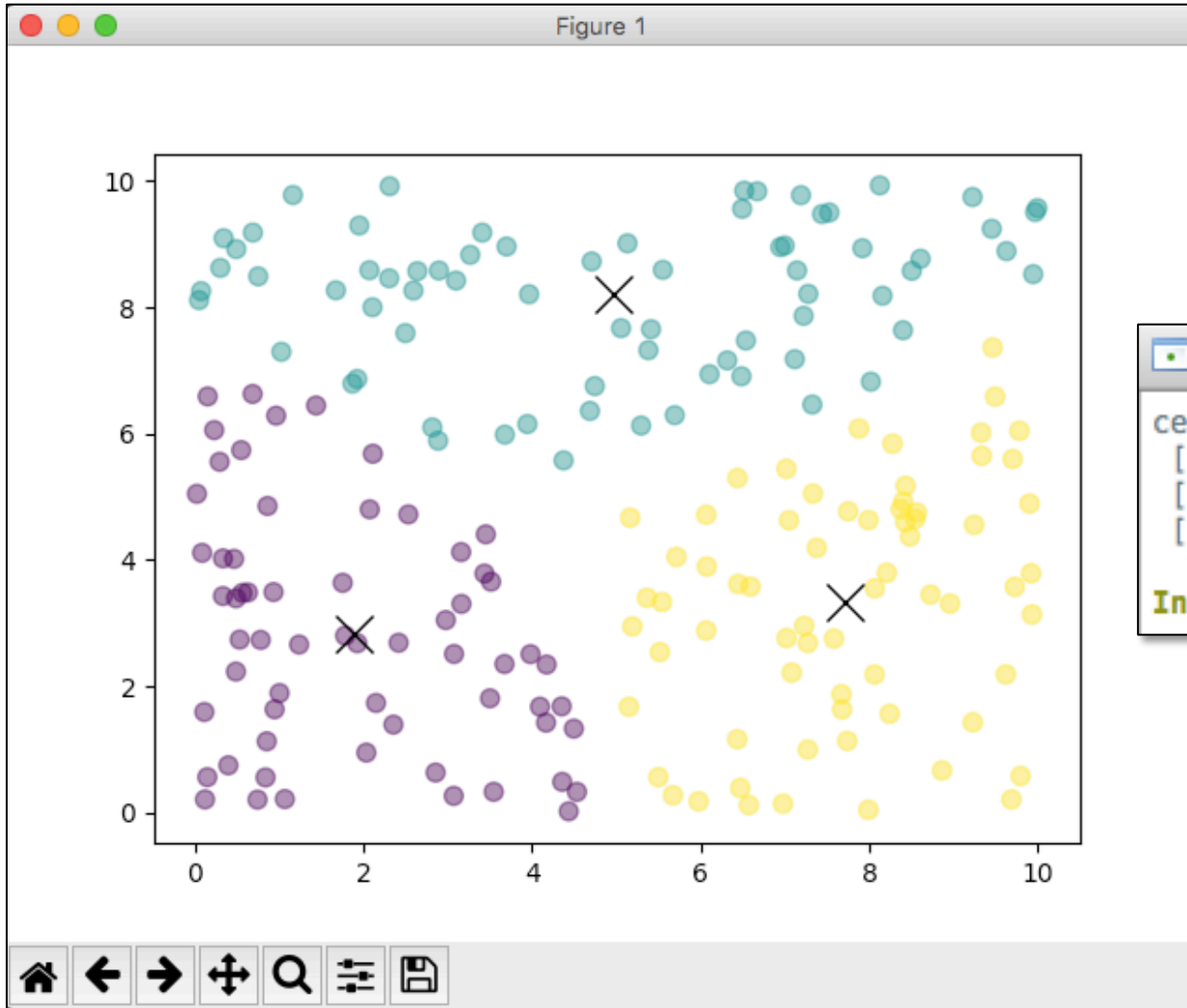
# K-means with TensorFlow

- Example:

```
49 # 5. Next we run the graph. For each iteration, we update the centroids and return
50 #   their values along with the cluster assignments values:
51 with tf.Session() as sess:
52     sess.run(init)
53     for step in range(iteration_n):
54         [_, centroid_values, points_values, assignment_values] = sess.run([update_centroids,
55                                     centroids, points, assignments])
56
57 # 6. Finally, we display the coordinates of the final centroids and a multi-colored
58 #   scatter plot showing how the data points have been clustered:
59 print("centroids" + "\n", centroid_values)
60 plt.scatter(points_values[:, 0], points_values[:, 1], c=assignment_values, s=50, alpha=0.5)
61 plt.plot(centroid_values[:, 0], centroid_values[:, 1], 'kx', markersize=15)
62 plt.pause(1)
63
64 # The data in a training set is grouped into clusters as the result of implementing
65 # the k-means algorithm in TensorFlow.
```

# K-means with TensorFlow

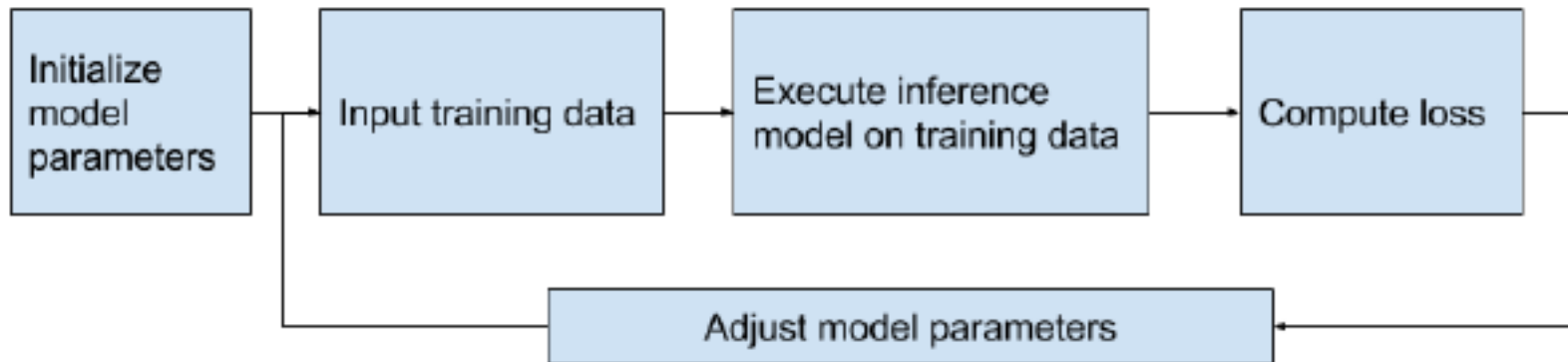
- Example:



```
Python ▾  
centroids  
[[5.37788891 2.41304675]  
 [7.27900734 7.46333703]  
 [1.97757837 6.70299535]]  
In [1]:
```

# Machine Learning

- The **models may vary** significantly in:
  - the number of operations they use (multiply, add, etc.)
  - the way they combine
  - the number of parameters they have
- Regardless, we **always apply the same general structure** for training them:





# Machine Learning

- We create a training loop that:
  - Initializes the model parameters for the first time
  - Reads the training data along with the expected output data for each data example
  - Executes the inference model on the training data, so it calculates for each training input example
  - Computes the loss
  - Adjusts the model parameters

# Machine Learning

- The **loop repeats this process** through a number of cycles, according to:
  - the **learning rate** that we need to apply, and
  - **depending on the model and data** we input to it.
- **After training, we apply an evaluation phase:**
  - we execute the inference against a different set data to which we also have the expected output, and evaluate the loss for it.

# Machine Learning

- Given how this dataset contains **examples unknown for the model**, the **evaluation** tells us how well the model predicts beyond its training.
- A very common practice is to **take the original dataset** and **randomly split** it in 70% of the examples for training, and 30% for evaluation
- Let's use this structure to define some generic frame for the model code:

# Machine Learning

```
import tensorflow as tf

# initialize variables/model parameters

# define the training loop operations
def inference(X):
    # compute inference model over data X and return the result

def loss(X, Y):
    # compute loss over training data X and expected outputs Y

def inputs():
    # read/generate input training data X and expected outputs Y

def train(total_loss):
    # train / adjust model parameters according to computed total loss

def evaluate(sess, X, Y):
    # evaluate the resulting trained model
```

# Machine Learning

```
# Launch the graph in a session, setup boilerplate
with tf.Session() as sess:

    tf.initialize_all_variables().run()

    X, Y = inputs()

    total_loss = loss(X, Y)
    train_op = train(total_loss)

    coord = tf.train.Coordinator()
    threads = tf.train.start_queue_runners(sess=sess, coord=coord)

    # actual training loop
    training_steps = 1000
    for step in range(training_steps):
        sess.run([train_op])
        # for debugging and learning purposes, see how the
        loss gets decremented thru training steps
```

# Machine Learning

```
        if step % 10 == 0:  
            print "loss: ", sess.run([total_loss])  
  
evaluate(sess, X, Y)  
  
coord.request_stop()  
coord.join(threads)  
sess.close()
```

# Saving training checkpoints

- As we already stated, training models implies **updating their parameters, or variables in Tensorflow, through many training cycles**
- **Variables are stored in memory**, so if the computer would lose power after many hours of training, we would lose all of that work, so ...
- We use `tf.train.Saver` class **to save the graph** variables in proprietary binary files
- **We should periodically save the variables**, create a checkpoint file, and eventually **restore the training from the most recent checkpoint** if needed.

# Saving training checkpoints

- In order to use the Saver we need to slightly change the training loop scaffolding code:

```
# model definition code ...

# Create a saver.
saver = tf.train.Saver()

# Launch the graph in a session, setup boilerplate
with tf.Session() as sess:

    # model setup....

    # actual training loop
    for step in range(training_steps):

        sess.run([train_op])

        if step % 1000 == 0:
            saver.save(sess, 'my-model', global_step=step)

    # evaluation...

    saver.save(sess, 'my-model', global_step=training_steps)

    sess.close()
```



# Saving training checkpoints

- If we wish to recover the training from a certain point, we should use the `tf.train.get_checkpoint_state()` method
- It will verify if we already have a checkpoint saved, and the `tf.train.Saver.restore()` method to recover the variable values

```
with tf.Session() as sess:

    # model setup....

    initial_step = 0

    # verify if we don't have a checkpoint saved already
    ckpt = tf.train.get_checkpoint_state(os.path.dir-
name(__file__))
    if ckpt and ckpt.model_checkpoint_path:
        # Restores from checkpoint
        saver.restore(sess, ckpt.model_checkpoint_path)
        initial_step =
int(ckpt.model_checkpoint_path.rsplit('-', 1)[1])

    #actual training loop
    for step in range(initial_step, training_steps):
        ...
```

# Linear models for classification

- **Binary** classification
- **Line separates** the two classes
  - Decision boundary - **defines where the decision changes** from one class value to the other
- **Prediction** is made by **plugging in observed values** of the attributes into the expression
  - Predict one class if output  $\geq 0$ , and the other class if output  $< 0$
- Boundary becomes a **high-dimensional plane** (*hyperplane*) when there are **multiple attributes**

# Linear Models

- Linear regression is the simplest form of modeling for a supervised learning problem
- Primarily used for **regression**:
  - Inputs (attribute values) and output are all numeric
- Output is the sum of the **weighted input** attribute **values**
- The trick is to **find good values for the weights**
- There are different ways of doing this, which we will consider later:
  - the most famous one is to **minimize the squared error**

# Example: Predicting CPU performance

- The table shows some data for which both the outcome and the attributes are numeric:

	Cycle time (ns)	Main memory (Kb)		Cache (Kb)	Channels		Performance
	MYCT	MMIN	MMAX	CACH	CHMIN	CHMAX	PRP
1	125	256	6000	256	16	128	198
2	29	8000	32000	32	8	32	269
...							
208	480	512	8000	32	0	0	67
209	480	1000	4000	0	0	0	45

# Example: Predicting CPU performance

- It concerns the relative performance of computer processing power on the basis of a number of relevant attributes:
  - each row represents one of 209 different computer configurations.

	Cycle time (ns)	Main memory (Kb)		Cache (Kb)	Channels		Performance
	MYCT	MMIN	MMAX	CACH	CHMIN	CHMAX	PRP
1	125	256	6000	256	16	128	198
2	29	8000	32000	32	8	32	269
...							
208	480	512	8000	32	0	0	67
209	480	1000	4000	0	0	0	45

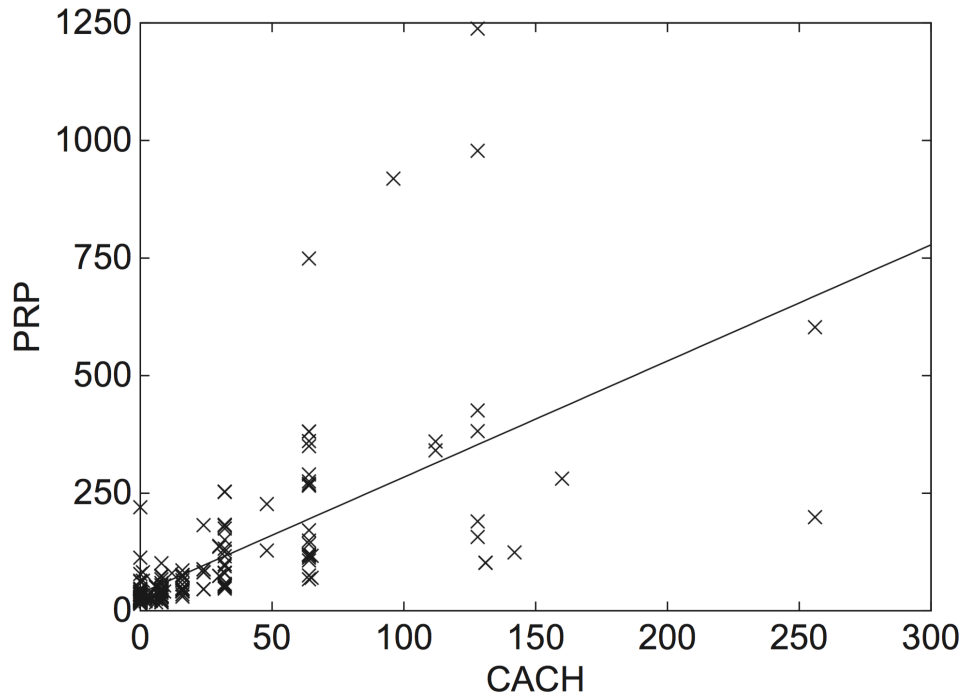
# Example: Predicting CPU performance

- The classic way of dealing with continuous prediction is to write the outcome as a linear sum of the attribute values with appropriate weights, in this *linear regression function*:

$$\text{PRP} = -55.9 + 0.0489 \text{ MYCT} + 0.0153 \text{ MMIN} + 0.0056 \text{ MMAX} + 0.6410 \text{ CACH} - 0.2700 \text{ CHMIN} + 1.480 \text{ CHMAX}$$

	Cycle time (ns)	Main memory (Kb)		Cache (Kb)	Channels		Performance
	MYCT	MMIN	MMAX	CACH	CHMIN	CHMAX	PRP
1	125	256	6000	256	16	128	198
2	29	8000	32000	32	8	32	269
...							
208	480	512	8000	32	0	0	67
209	480	1000	4000	0	0	0	45

# A linear regression function for the CPU performance data



$$\text{PRP} = 37.06 + 2.47\text{CACH}$$

- Given a set of data points as training data, you are going to **find the linear function that best fits** them.
- In a **2-dimensional dataset**, this type of function represents a **straight line**.
- The **x-marks are the training data points** and the **line is the what the model will infer (predict)**.

# TensorFlow

- Working with **Linear regression**
  - What is Linear regression?
    - It is a standard technique for numeric prediction
    - It is the simplest form of modeling for a supervised learning problem
    - The outcome is linear combination of attributes
$$x = w_0 + w_1a_1 + w_2a_2 + \dots + w_ka_k$$
    - Weights are calculated from the training data
    - Predicted value for first training instance  $\mathbf{a}^{(1)}$

$$w_0a_0^{(1)} + w_1a_1^{(1)} + w_2a_2^{(1)} + \dots + w_ka_k^{(1)} = \sum_{j=0}^k w_ja_j^{(1)}$$

(assuming each instance is extended with a constant attribute with value 1)



# TensorFlow

- Working with **Linear regression**
  - Another general representation of a linear function is:

$$y(x_1, x_2, \dots, x_k) = w_1x_1 + w_2x_2 + \dots + w_kx_k + b$$

- And its matrix (or tensor) form is:

$$Y = XW^T + b \text{ where } X = (x_1, \dots, x_k) \quad W = (w_1, \dots, w_k)$$

- $Y$  is the value we are trying to predict.
- $x_1, \dots, x_k$  independent or predictor variables are the values that we provide when using our model for predicting new values. In matrix form, you can provide multiple examples at once- one per row.
- $w_1, \dots, w_k$  are the parameters the model will learn from the training data, or the “weights” given to each variable.
- $b$  is also a learned parameter- the constant of the linear function that is also known as the bias of the model.

# TensorFlow

- Instead of transposing weights, we can simply define them as a single column vector
- Here is a sample model:

```
# initialize variables/model parameters
W = tf.Variable(tf.zeros([2, 1]), name="weights")
b = tf.Variable(0., name="bias")

def inference(X):
    return tf.matmul(X, W) + b
```

# TensorFlow

- Now we have to define how to **compute the loss**.
- For a simple model we can use the **squared error**:
  - It sums the squared difference of all the predicted values for each training example with their corresponding expected values.
  - Algebraically it is the squared Euclidean distance between the predicted output vector and the expected one.
- Graphically **in a 2d dataset the error is the length of the vertical line** that you can trace from the expected data point to the predicted regression line.

# TensorFlow

- It is also known as L2 norm or L2 loss function.
- We use it squared to avoid computing the square root, since it makes no difference for trying to minimize the loss and saves a computing step:

$$loss = \sum_i (y_i - y_{predicted_i})^2$$

- We sum over  $i$ , where  $i$  is each data example. In code:

```
def loss(X, Y):  
    Y_predicted = inference(X)  
    return tf.reduce_sum(tf.squared_difference(Y, Y_predicted))
```

# TensorFlow

- Let's train our model with data:

```
def inputs():  
    weight_age = [[84, 46], [73, 20], [65, 52], [70, 30], [76,  
57], [69, 25], [63, 28], [72, 36], [79, 57], [75, 44], [27,  
24], [89, 31], [65, 52], [57, 23], [59, 60], [69, 48], [60,  
34], [79, 51], [75, 50], [82, 34], [59, 46], [67, 23], [85,  
37], [55, 40], [63, 30]]  
    blood_fat_content = [354, 190, 405, 263, 451, 302, 288,  
385, 402, 365, 209, 290, 346, 254, 395, 434, 220, 374, 308,  
220, 311, 181, 274, 303, 244]  
  
    return tf.to_float(weight_age), tf.to_float(blood_fat_con-  
tent)
```

Data source: <http://people.sc.fsu.edu/~jburkardt/datasets/regression/x09.txt>

# TensorFlow

- Next we define the model training operation.
- We will use the gradient descent algorithm for optimizing the model parameters:

```
def train(total_loss):  
    learning_rate = 0.0000001  
    return tf.train.GradientDescentOptimizer(learning_rate).minimize(total_loss)
```

- When you run it, you will see printed how the loss gets smaller on each training step.

# TensorFlow

- Now that we trained the model, it's time to evaluate it:

```
def evaluate(sess, X, Y):  
    print sess.run(inference([[80., 25.]))) # ~ 303  
    print sess.run(inference([[65., 25.]))) # ~ 256
```

- As a quick evaluation, you can check that the model learned how the blood fat decays with weight
- Also, the output values are in between the boundaries of the original trained values.

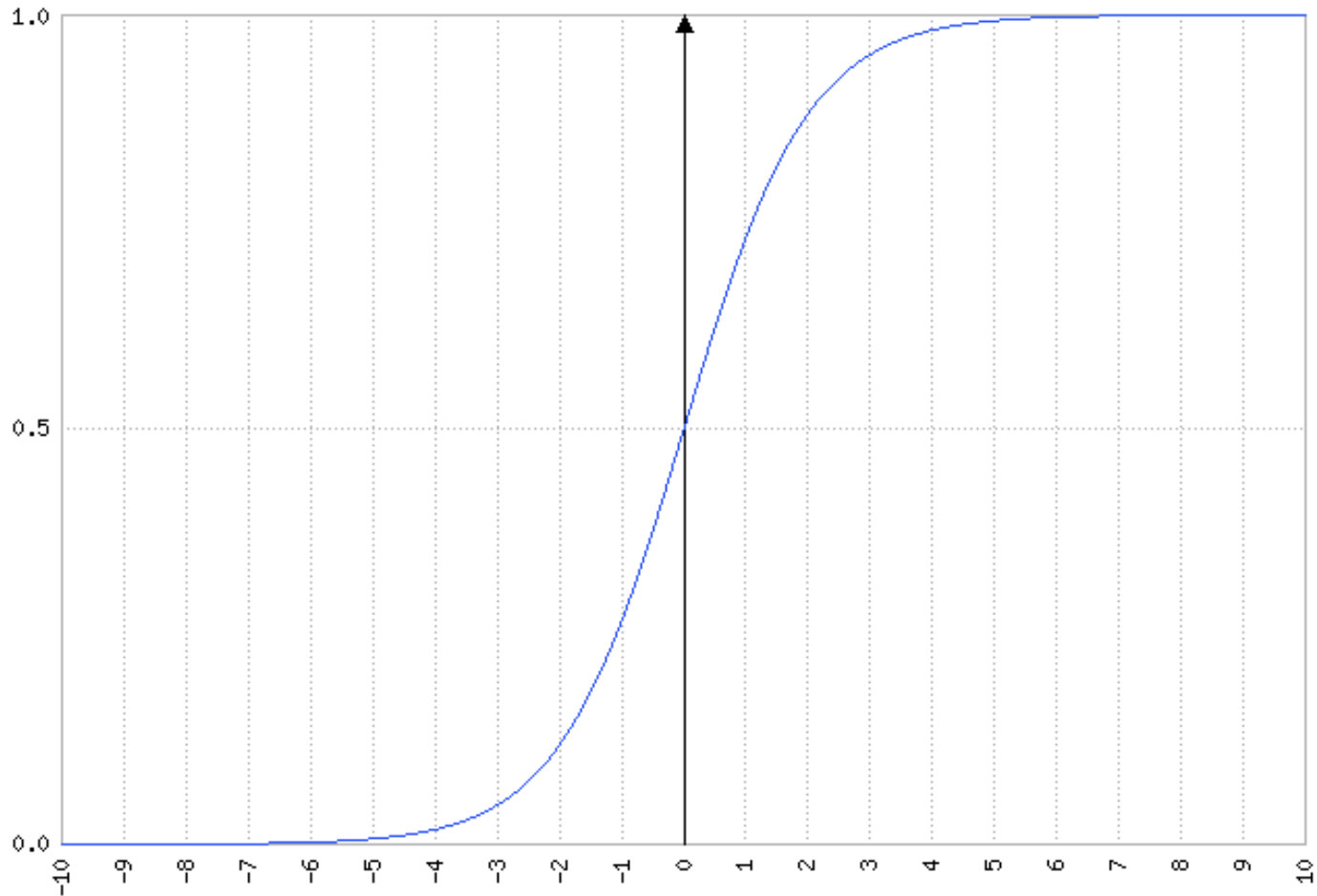
# TensorFlow

- Working with **Logistic regression**
  - The linear regression model predicts a continuous value, or any real number.
  - Now we are going to present a model that can answer a **yes-no** type of question, like “Is this email spam?”
- There is a function used commonly in machine learning called the **logistic function**. It is also known as the *sigmoid function*, because its shape is an S (and sigma is the Greek letter equivalent to s).

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



# TensorFlow



# TensorFlow

- In order to feed the function with the multiple dimensions, or features from the examples of our training datasets, we need to combine them into a single value

```
# same params and variables initialization as log reg.
W = tf.Variable(tf.zeros([5, 1]), name="weights")
b = tf.Variable(0., name="bias")

# former inference is now used for combining inputs
def combine_inputs(X):
    return tf.matmul(X, W) + b

# new inferred value is the sigmoid applied to the former
def inference(X):
    return tf.sigmoid(combine_inputs(X))
```

# TensorFlow

- Consider an example where the expected answer is “yes” and the model is predicting a very low probability for it, close to 0.
- This means that it is close to 100% hence the answer is “no”
- The **squared error penalizes such a case** with the same order of magnitude for the loss as if the probability would have been predicted as 20, 30, or even 50% for the “no” output
- There is a loss function that works better for this type of problem, which is the **cross entropy** function

$$loss = - \sum_i (y_i \cdot \log(y\_predicted_i) + (1 - y_i) \cdot \log(1 - y\_predicted_i))$$

# Criterion for attribute selection

- Which is the best attribute?
  - Want to get the **smallest tree**
  - **Heuristic**: choose the attribute that produces the “purest” nodes
- Popular selection criteria: **information gain**
  - Information gain **increases with** the average **purity** of the subsets
- Strategy: amongst attributes available **for splitting, choose attribute that gives greatest information gain**
- Information gain requires measure of **impurity**
- **Impurity** measure that it **uses** is the **entropy** of the class distribution, which is a measure from information theory

# What is Entropy?

- *Various definitions same meaning:*
  - *Entropy*: lack of order or predictability; gradual decline into disorder
  - *Entropy*: (in information theory) a logarithmic measure of the rate of transfer of information in a particular message or language
  - *Entropy* may be understood as a measure of disorder within a macroscopic system
  - *Entropy* is the measure of the level of disorder in a closed but changing system, a system in which energy can only be transferred in one direction from an ordered state to a disordered state
  - The higher the entropy, the higher the disorder and the system's energy to do useful work is lower

# Computing information

- We have a probability distribution: the class distribution in a subset of instances
- The **expected information** required to determine an outcome (i.e., class value), **is the distribution's *entropy***
- Formula for computing the entropy:
$$\text{Entropy}(p_1, p_2, \dots, p_n) = -p_1 \log p_1 - p_2 \log p_2 \dots - p_n \log p_n$$
- Using **base-2 logarithms**, entropy gives the information required in expected ***bits***
- ***Entropy is maximal when all classes are equally likely and minimal when one of the classes has probability 1***

# Example: attribute *Outlook*

- *Outlook = Sunny* :

$$\text{Info}([2, 3]) = 0.971 \text{ bits}$$

$$\text{Entropy}(S_{\text{sunny}}) = -\frac{2}{5} \log_2 \left( \frac{2}{5} \right) - \frac{3}{5} \log_2 \left( \frac{3}{5} \right)$$

- *Outlook = Overcast* :

$$\text{Info}([4, 0]) = 0.0 \text{ bits}$$

$$\text{Entropy}(S_{\text{overcast}}) = -\frac{4}{4} \log_2 \left( \frac{4}{4} \right) - 0 \log_2 (0)$$

- *Outlook = Rainy* :

$$\text{Info}([3, 2]) = 0.971 \text{ bits}$$

$$\text{Entropy}(S_{\text{rain}}) = -\frac{3}{5} \log_2 \left( \frac{3}{5} \right) - \frac{2}{5} \log_2 \left( \frac{2}{5} \right)$$

- Expected information for attribute:

$$\begin{aligned} \text{Info}([2, 3], [4, 0], [3, 2]) &= (5/14) \times 0.971 + (4/14) \times 0 + (5/14) \times 0.971 \\ &= 0.693 \text{ bits} \end{aligned}$$

# Computing information gain

- Information gain: information before splitting – information after splitting

$$\begin{aligned}\text{Gain ( Outlook )} &= \text{Info}([9,5]) - \text{info}([2,3],[4,0],[3,2]) \\ &= 0.940 - 0.693 \\ &= 0.247 \text{ bits}\end{aligned}$$

- Information gain for attributes from weather data:

$$\text{Gain ( Outlook )} = 0.247 \text{ bits}$$

$$\text{Gain ( Temperature )} = 0.029 \text{ bits}$$

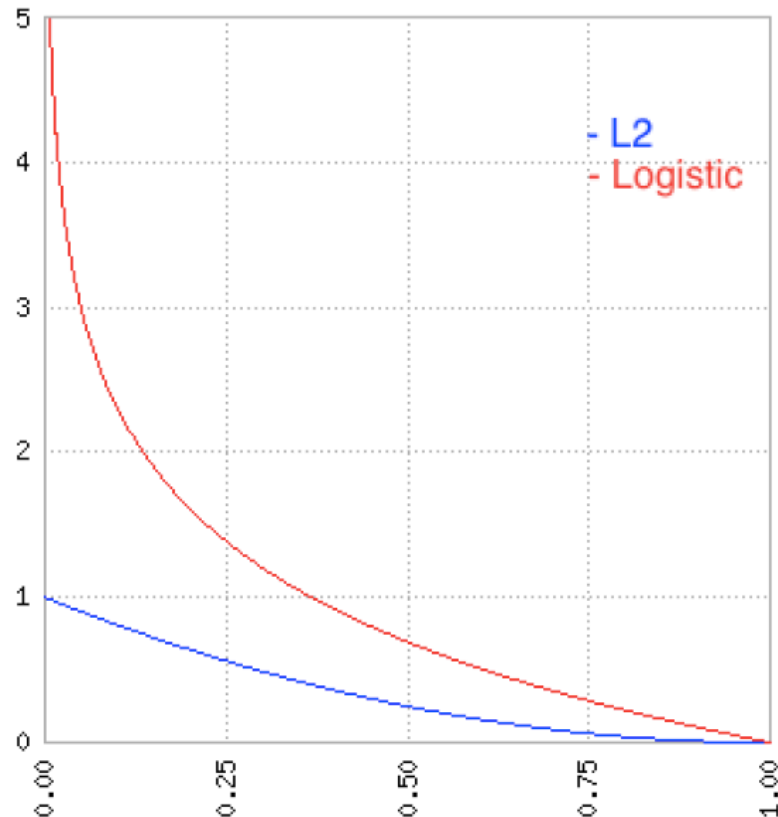
$$\text{Gain ( Humidity )} = 0.152 \text{ bits}$$

$$\text{Gain ( Windy )} = 0.048 \text{ bits}$$



# TensorFlow

- We can visually compare the behavior of both loss functions according to the predicted output for a “yes.”



# TensorFlow

- There is a Tensorflow method for calculating cross entropy directly for a sigmoid output in a single, optimized step:

```
def loss(X, Y):  
    return tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_log-  
its(combine_inputs(X), Y))
```

- **Cross entropy** meaning:
  - *In information theory, Shannon entropy allows to estimate the average minimum number of bits needed to encode a symbol  $s_i$  from a string of symbols, based on the probability  $p_i$  of each symbol to appear in that string.*

$$H = - \sum_i (p_i \cdot \log_2(p_i))$$

# TensorFlow

- **Cross entropy** example:

- *Task*: calculate the entropy for the word “HELLO.”

- *Solution*:

$$p(H) = p(E) = p(O) = 1/5 = 0.2$$

$$p(L) = 2/5 = 0.4$$

$$H = -3 \cdot 0.2 \cdot \log_2(0.2) - 0.4 \cdot \log_2(0.4) = 1.92193$$

- *Conclusion*: we need 2 bits per symbol to encode “HELLO” in the optimal encoding.

# TensorFlow

- Let's apply the model to real data using the Titanic survivor contest dataset from:

*<https://www.kaggle.com/c/titanic/data>*

- We need to load the data first, so download the **train.csv** file
- You can load and parse it, and create a batch to read many rows packed in a single tensor for computing the inference efficiently: ... *code on next slide*

# TensorFlow

```
def read_csv(batch_size, file_name, record_defaults):
    filename_queue = tf.train.string_input_producer([
        os.path.join(os.getcwd(), file_name)])

    reader = tf.TextLineReader(skip_header_lines=1)
    key, value = reader.read(filename_queue)

    # decode_csv will convert a Tensor from type string (the
    text line) in
    # a tuple of tensor columns with the specified defaults,
    which also
    # sets the data type for each column
    decoded = tf.decode_csv(value, record_defaults=record_defaults)

    # batch actually reads the file and loads "batch_size"
    rows in a single tensor
    return tf.train.shuffle_batch(decoded,
                                   batch_size=batch_size,
                                   capacity=batch_size * 50,
                                   min_after_dequeue=batch_size)
```

# TensorFlow

- The model will have to infer, based on the passenger *age*, *sex* and *ticket* class if the passenger **survived** or **not**
- We have to use **categorical data** from this dataset:
  - Ticket class and gender are string features with a predefined possible set of values that they can take
  - To use them in the inference model we need to convert them to numbers
  - A naive approach might be assigning a number for each possible value
  - For instance, you can use “1” for first ticket class, “2” for second, and “3” for third

# TensorFlow

- When working with categorical data, convert it to multiple boolean features, one for each possible value.
- This allows the model to weight each possible value separately.
- In the case of categories with only two possible values, like the gender in the dataset, it is enough to have a single variable for it.
- That's because you can express a linear relationship between the values.
  - For instance if possible values are **female = 1** and **male = 0**, then **male = 1 - female**, a single weight can learn to represent both possible states.

# TensorFlow

```
def inputs():
    passenger_id, survived, pclass, name, sex, age, sibsp,
    parch, ticket, fare, cabin, embarked = \
        read_csv(100, "train.csv", [[0.0], [0.0], [0], [""],
[""], [0.0], [0.0], [0.0], [""], [0.0], [""], [""]])

    # convert categorical data
    is_first_class = tf.to_float(tf.equal(pclass, [1]))
    is_second_class = tf.to_float(tf.equal(pclass, [2]))
    is_third_class = tf.to_float(tf.equal(pclass, [3]))

    gender = tf.to_float(tf.equal(sex, ["female"]))

    # Finally we pack all the features in a single matrix;
    # We then transpose to have a matrix with one example per
    row and one feature per column.
    features = tf.transpose(tf.pack([is_first_class, is_sec-
ond_class, is_third_class, gender, age]))
    survived = tf.reshape(survived, [100, 1])

    return features, survived
```



# TensorFlow

- Lets train our model now:

```
def train(total_loss):  
    learning_rate = 0.01  
    return tf.train.GradientDescentOptimizer(learning_rate).minimize(total_loss)
```

- To evaluate the results we are going to run the inference against a batch of the training set and count the number of examples that were correctly predicted. We call that measuring the **accuracy**

```
def evaluate(sess, X, Y):  
  
    predicted = tf.cast(inference(X) > 0.5, tf.float32)  
  
    print sess.run(tf.reduce_mean(tf.cast(tf.equal(predicted,  
Y), tf.float32)))
```

# Softmax classification

- With logistic regression we were able to model the answer to the yes-no question.
- Now we want to be able to answer a multiple choice type of question like: “Were you born in Boston, London, or Sydney?”
- For that case there is the **softmax** function, which is a generalization of the logistic regression for C possible different values:

$$f(x)_c = \frac{e^{-x_c}}{\sum_{j=0}^{C-1} e^{-x_j}} \text{ for } c = 0 \dots C - 1$$

# Softmax classification

- **Softmax** function returns a probability vector of  $C$  components, filling the corresponding probability for each output class
- The sum of the  $C$  vector components always = 1 (*it is a probability*)
- To code this model, **we will need to change** from the previous models in the variable initialization
- Given that our model computes  $C$  outputs instead of just one, **we need to have  $C$  different weight groups**, one for each possible output
- **So, we will use a weights matrix, instead of a weights vector**
- That matrix will have one row for each input feature, and one column for each output class

# Softmax classification

- Let's use the classical Iris flower dataset for trying softmax
- You can download it from here:

*<https://archive.ics.uci.edu/ml/datasets/Iris>*

- The set contains:
  - 4 data features and
  - 3 possible output classes, one for each type of iris plant
  - so our weights matrix should have a 4x3 dimension

# Softmax classification

- The variable initialization code should look like this:

```
# this time weights form a matrix, not a vector, with one "feature weights column" per output class.  
W = tf.Variable(tf.zeros([4, 3]), name="weights")  
# so do the biases, one per output class.  
b = tf.Variable(tf.zeros([3], name="bias"))
```

- Tensorflow contains an embedded implementation of the **softmax** function:

```
def inference(X):  
    return tf.nn.softmax(combine_inputs(X))
```

# Softmax classification

- For a single training example  $i$ , cross entropy now becomes:

$$loss_i = - \sum_c y_c \cdot \log(y\_predicted_c)$$

*We are summing the loss for each output class on that training example*

- Next, to calculate the total loss among the training set, we sum the loss for each training example:

$$loss = - \sum_i \sum_c y_{c_i} \cdot \log(y\_predicted_{c_i})$$

# Softmax classification

- There are **two** versions implemented in Tensorflow for the softmax crossentropy function: *one specially optimized for training sets with a single class value per example*
- For example, our training data may have a class value that could be either “dog”, “person” or “tree”.
- That function is:

*tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits*

```
def loss(X, Y):  
    return tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(combine_inputs(X), Y))
```

# Softmax classification

- There are **two** versions implemented in Tensorflow for the softmax crossentropy function: *the other, lets you work with training sets containing the probabilities of each example to belong to every class*
- For example, you could use training data like “60% of the asked people consider that this picture is about dogs, 25% about trees, and the rest about a person”.
- That function is:  
*tf.nn.softmax\_cross\_entropy\_with\_logits*



# Softmax classification

- Let's define our input method. We will reuse the `read_csv` function from the `logistic regression` example, but will call it with the defaults for the values on our dataset, which are all numeric:

```
def inputs():  
    sepal_length, sepal_width, petal_length, petal_width, label =\  
        read_csv(100, "iris.data", [[0.0], [0.0], [0.0],  
        [0.0], [""]])
```

# Softmax classification

```
# convert class names to a 0 based class index.
label_number = tf.to_int32(tf.argmax(tf.to_int32(tf.pack([
    tf.equal(label, ["Iris-setosa"]),
    tf.equal(label, ["Iris-versicolor"]),
    tf.equal(label, ["Iris-virginica"])
])), 0))

# Pack all the features that we care about in a single matrix;
# We then transpose to have a matrix with one example per
row and one feature per column.
features = tf.transpose(tf.pack([sepal_length, sepal_width, petal_length, petal_width]))

return features, label_number
```

# Softmax classification

- The training function is also the same
- For evaluation of accuracy, we need a slight change from the sigmoid version:

```
def evaluate(sess, X, Y):  
    predicted = tf.cast(tf.argmax(inference(X), 1), tf.int32)  
  
    print sess.run(tf.reduce_mean(tf.cast(tf.equal(predicted,  
Y), tf.float32)))
```

- Running the code should print an accuracy of about 96%.

# TensorFlow

- HW2:

*Recreate the graph and visualize it in Tensorboard using:*

- 1. Placeholder for an input array with dtype float32 and shape None*
- 2. Scopes for the input, middle section and final node f*
- 3. Feed the placeholder with an array A consisting of 100 normally distributed random numbers with Mean = 1 and Standard deviation = 2*
- 4. Save your graph and show it in TensorBoard*
- 5. Plot you input array on a separate figure*
- 6. Make sure you comment your code well and provide your name on top of your work*
- 7. Email your Github link (or code directly) to me including your .py file + screenshots of TensorBoard*

