CS 696 Intro to Big Data: Tools and Methods
Spring Semester, 2019
Doc 18 Regression, Transformation
Apr 9, 2019

Agile Software Development

Iterative and incremental software development

Scrum 1995

Crystal Clear 1996

Extreme programming (XP) 1996

Feature-driven development 1997

Manifesto for Agile Software Development

2001

Value

Individuals and Interactions more than processes and tools

Working Software more than comprehensive documentation

Customer Collaboration more than contract negotiation

Responding to Change more than following a plan

Agile software development principles

Customer satisfaction by early and continuous delivery of valuable software

Welcome changing requirements, even in late development

Working software is delivered frequently (weeks rather than months)

Close, daily cooperation between business people and developers

Projects are built around motivated individuals, who should be trusted

Face-to-face conversation is the best form of communication (co-location)

Working software is the primary measure of progress

Sustainable development, able to maintain a constant pace

Continuous attention to technical excellence and good design

Simplicity—the art of maximizing the amount of work not done—is essential

Best architectures, requirements, and designs emerge from self-organizing teams

Regularly, the team reflects on how to become more effective, and adjusts accordingly

A manifesto for Agile data science

https://www.oreilly.com/ideas/a-manifesto-for-agile-data-science

http://tinyurl.com/y8lzavxn

The manifesto focuses on how to think

The key is that you approach data science in an active and dynamic way

A manifesto for Agile data science

Iterate, iterate, iterate

Insight comes from the 25th query in a chain of queries, not the first one

Data tables have to be parsed, formatted, sorted, aggregated, and summarized before they can be understood

A manifesto for Agile data science

Ship intermediate output

Often left at the end of a sprint with things that aren't complete

In Agile data science, we document and share the incomplete assets we create as we work

We commit all work to source control

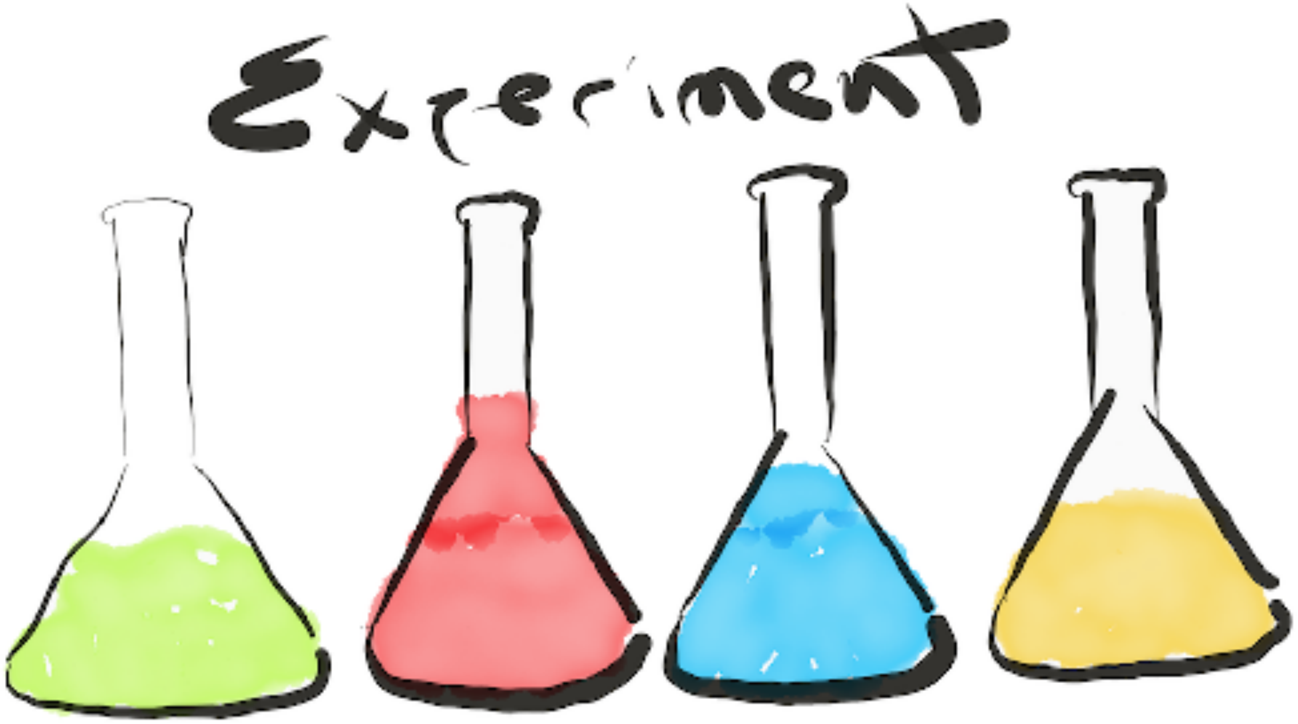
We share this work with teammates and, as soon as possible, with end users

A manifesto for Agile data science

Perform experiments, not tasks

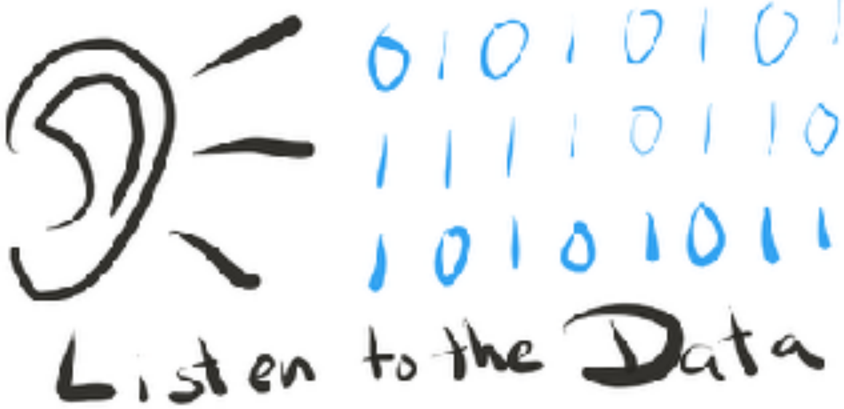
Data science differs from software engineering in that it is part science, part engineering

In any given task, we must iterate to achieve insight, and these iterations can best be summarized as experiments



A manifesto for Agile data science

Listen to the data



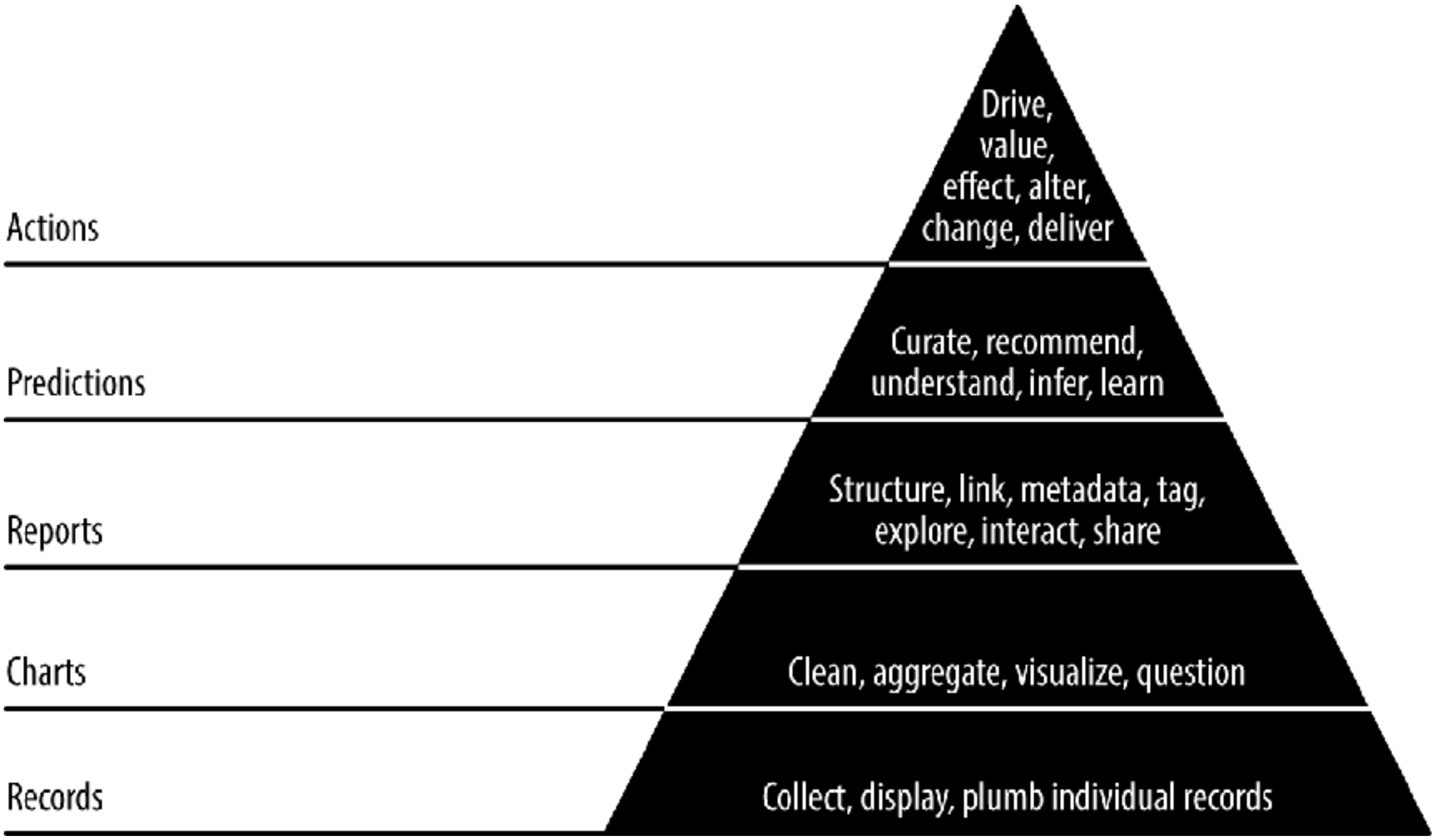
What is possible is as important as what is intended

Without understanding what the data “has to say” about any feature, the product owner can’t do a good job

The data’s opinion must always be included in product discussions, which means that they must be grounded in **visualization** through exploratory data analysis in the internal application that becomes the focus of our efforts

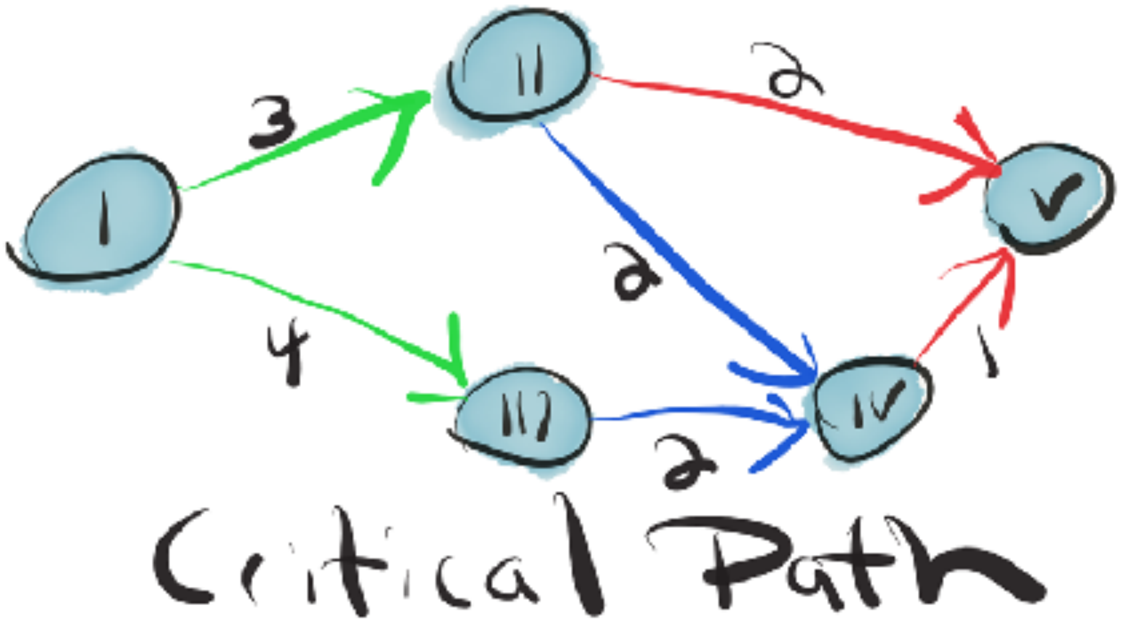
A manifesto for Agile data science

Respect the data-value pyramid


Actions
Drive, value, effect, alter, change, deliver
Predictions
Curate, recommend, understand, infer, learn
Reports
Structure, link, metadata, tag, explore, interact, share
Charts
Clean, aggregate, visualize, question
Records
Collect, display, plumb individual records

A manifesto for Agile data science

Find the critical path



To maximize our odds of success, we should focus most of our time on that aspect of our application that is most essential to its success. But which aspect is that? This must be discovered through experimentation. Analytics product development is the search for and pursuit of a moving goal.

A manifesto for Agile data science

Get meta

The focus is on documenting the analytics process as opposed to the end state or product we are seeking. This lets us be Agile and ship intermediate content as we iteratively climb the data-value pyramid to pursue the critical path to a killer product.

Review

org.apache.spark.**ml** vs org.apache.spark.**mllib**

Normalizing Data

Linear Regression
- Fitting data to line
- Residuals
- Pearson’s Correlation

Computing Pearson's Correlation r

f(x) = x

f(x)	x
1	1
2	2
3	3
5	5
10	10
20	20
30	30
100	100
500	500

linearExactSimple.csv

```
y,x
1,1
2,2
3,3
5,5
10,10
20,20
30,30
100,100
500,500
```

```
linear = spark.read.format("csv").
        option("header",true).
        option("inferschema",true).
        load("linearExactSimple.csv")
linear.show


linear.stat.corr("y","x")

        1.0
```

```
+---+---+
|  y|  x|
+---+---+
|  1|  1|
|  2|  2|
|  3|  3|
|  5|  5|
| 10| 10|
| 20| 20|
| 30| 30|
|100|100|
|500|500|
+---+---+
from pyspark.sql.functions import lit
withOnes = linear.withColumn("1",lit(1))
withOnes.show()
```

```
+---+---+---+
|  y|  x|  1|
+---+---+---+
|  1|  1|  1|
|  2|  2|  1|
|  3|  3|  1|
|  5|  5|  1|
| 10| 10|  1|
| 20| 20|  1|
| 30| 30|  1|
|100|100|  1|
|500|500|  1|
+---+---+---+
```

```
withOnes.stat.corr("y","1")
```

```
NaN
```

Regression Example

f(x) = x

f(x)	x
1	1
2	2
3	3
5	5
10	10
20	20
30	30
100	100
500	500

Regression model requires

label (dependent variable) - Double

features (independent variable) - Vector of doubles

- Use
  - SVM format
  - Transformers

SVM, Libsvm, File format

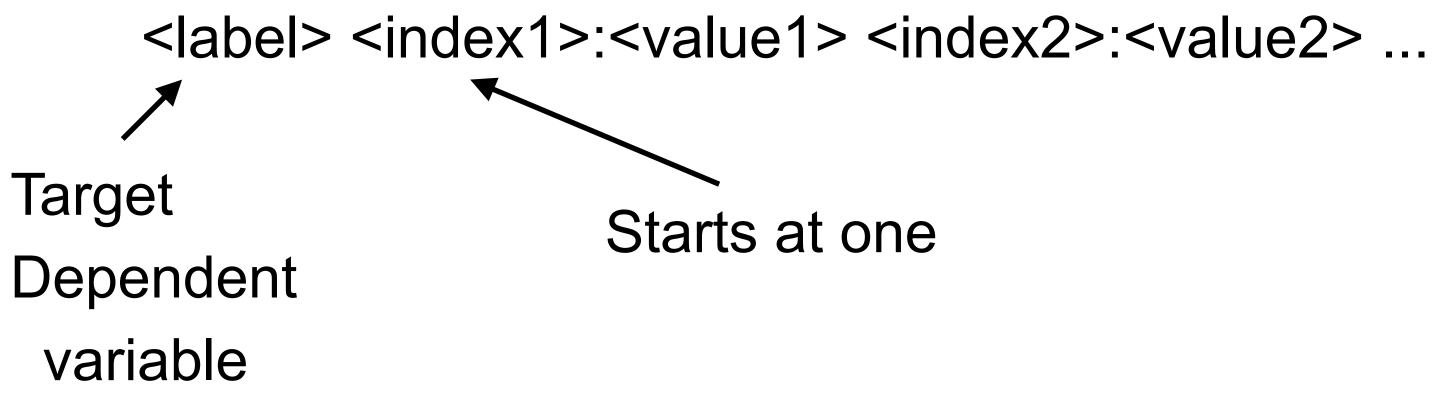
SVM - Support Vector Machines

- Supervised learning models with learning algorithms
- Classification & regression

LIBSVM

- Popular machine learning library
- National Taiwan University
- Open source
- Code reused in other open source machine learning toolkits
    - scikit

File Format



<label> <index1>:<value1> <index2>:<value2> ...

Target
Dependent
variable

Starts at one

Linear Regression Example

f(x) = x

Training Data
linearExact.csv

```
1 1:1
2 1:2
3 1:3
5 1:5
10 1:10
20 1:20
30 1:30
100 1:100
500 1:500
```

label
f(x)

features
x

Test Data
numbers.txt

```
1 1:-50
2 1:-10
3 1:50
4 1:75
5 1:1000
```

Not
Used

features
x

Basic Process

org.apache.spark.ml.regression.LinearRegression

Read Training data

Create LinearRegression object

Fit LinearRegression object to training data to get linear regression model

Read data

Evaluate data using linear regression model

Reading the Data

```
training = spark.read.format("libsvm").load("linearExact.svm")
training.show()
```

```
+-----+---------------+
|label|       features|
+-----+---------------+
|  1.0|  (1,[0],[1.0])|
|  2.0|  (1,[0],[2.0])|
|  3.0|  (1,[0],[3.0])|
|  5.0|  (1,[0],[5.0])|
| 10.0| (1,[0],[10.0])|
| 20.0| (1,[0],[20.0])|
| 30.0| (1,[0],[30.0])|
|100.0|(1,[0],[100.0])|
|500.0|(1,[0],[500.0])|
+-----+---------------+
```

linearExact.svm

```
1 1:1
2 1:2
3 1:3
5 1:5
10 1:10
20 1:20
30 1:30
100 1:100
500 1:500
```

Fitting the Data

```
from pyspark.ml.regression import LinearRegression

training = spark.read.format("libsvm").load("linearExact.svm")

linearRegression = LinearRegression().setMaxIter(10)

lrModel = linearRegression.fit(training)

print("Coefficients: " + str(lrModel.coefficients) + " Intercept: " + str(lrModel.intercept))
```

```
Coefficients: [0.9999999999999998] Intercept: 2.1042353059790043E-14
```

Measuring the Model

```
trainingSummary = lrModel.summary
trainingSummary.residuals.show()

print("RMSE: " + str(trainingSummary.rootMeanSquaredError))
print("r2: " + str(trainingSummary.r2))
```

```
+--------------------+
|           residuals|
+--------------------+
|-2.08721928629529...|
|-2.04281036531028...|
|-1.99840144432528...|
|-2.04281036531028...|
|-1.95399252334027...|
|-1.77635683940025...|
|-1.42108547152020...|
|1.421085471520200...|
|1.136868377216160...|
+--------------------+

RMSE: 4.176065532788523E-14
r2: 1.0
```

Using The Model

Test Data
numbers.txt

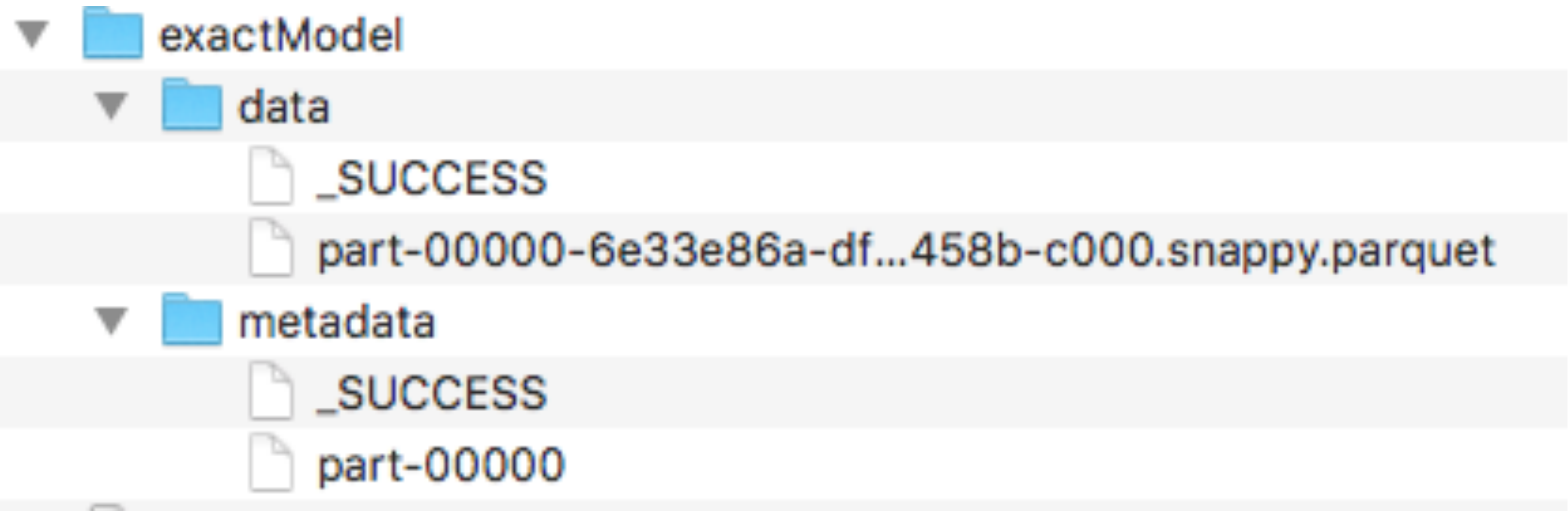
```
data = spark.read.format("libsvm").load("numbers.svm")

summary  = lrModel.evaluate(data)
predictions = summary.predictions
predictions.show()
```

```
1 1:-50
2 1:-10
3 1:50
4 1:75
5 1:1000
```

```
+-----+----------------+-------------------+
|label|        features|         prediction|
+-----+----------------+-------------------+
|  1.0| (1,[0],[-50.0])|-49.999999999999964|
|  2.0| (1,[0],[-10.0])| -9.999999999999977|
|  3.0|  (1,[0],[50.0])|  50.00000000000001|
|  4.0|  (1,[0],[75.0])|               75.0|
|  5.0|(1,[0],[1000.0])|  999.9999999999998|
+-----+----------------+-------------------+
```

Saving Model

lrModel.save('exactModel')



```
exactModel
    data
        _SUCCESS
        part-00000-6e33e86a-df...458b-c000.snappy.parquet
    metadata
        _SUCCESS
        part-00000
```

Entire Program

```
from pyspark.ml.regression import LinearRegression

training = spark.read.format("libsvm").load("linearExact.svm")

linearRegression = LinearRegression().setMaxIter(10)
lrModel = linearRegression.fit(training)

data = spark.read.format("libsvm").load("numbers.svm")

summary  = lrModel.evaluate(data)
predictions = summary.predictions
predictions.show()

lrModel.save('exactModel')
```

Reusing the Model

```
from pyspark.ml.regression import LinearRegressionModel

lrModel2 = LinearRegressionModel.load("exactModel")

data = spark.read.format("libsvm").load("numbers.svm")
summary = lrModel2.evaluate(data)
summary.predictions.show()
```

But What if data is not in SVM format?

linearExactSimple.csv

```
y,x
1,1
2,2
3,3
5,5
10,10
20,20
30,30
100,100
500,500
```

Use Transformers

- RFormula
- VectorAssembler

VectorAssembler

A feature transformer that merges multiple columns into a vector column of double

Create VectorAssembler

set input columns

set output column

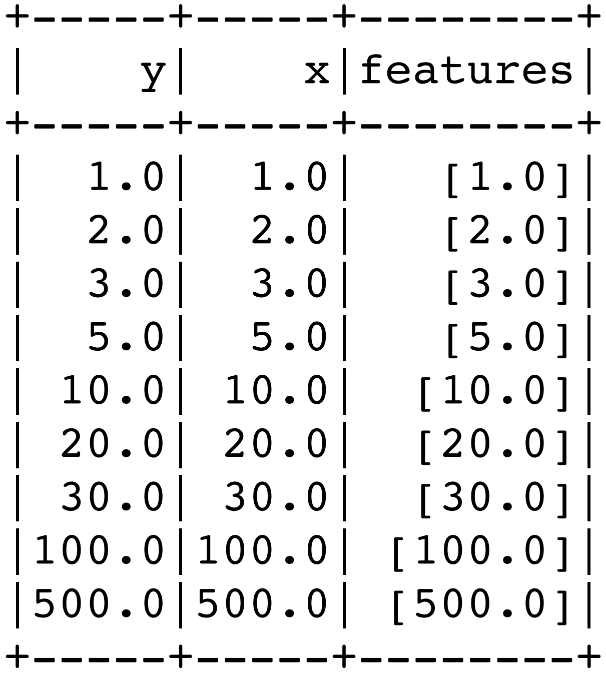
Transform dataFrame

```
data = spark.read.format("csv"). \
       option("header",True).\
       option("inferschema",True).\
       load("linearExactSimple.csv")

from pyspark.ml.feature import VectorAssembler

assembler =  VectorAssembler(inputCols=["x"], outputCol="features")
result = assembler.transform(data)
result.show()
```

linearExactSimple.csv

```
y,x
1,1
2,2
3,3
5,5
10,10
20,20
30,30
100,100
500,500
```

result



```
+-----+-----+--------+
|    y|    x|features|
+-----+-----+--------+
|  1.0|  1.0|   [1.0]|
|  2.0|  2.0|   [2.0]|
|  3.0|  3.0|   [3.0]|
|  5.0|  5.0|   [5.0]|
| 10.0| 10.0|  [10.0]|
| 20.0| 20.0|  [20.0]|
| 30.0| 30.0|  [30.0]|
|100.0|100.0| [100.0]|
|500.0|500.0| [500.0]|
+-----+-----+--------+
```

More than one Independent variable

twoVariables.csv

$y = 2x + z + 5*\mathrm{rand}()$

$-1 \le \mathrm{rand}() \le 1$

```
y,x,z
11.26,1,1
9.67,1,2
17.77,2,3
11.55,2,4
22.18,3,5
17.78,3,6
19.66,4,7
27.50,4,8
28.96,5,9
28.53,5,10
25.45,6,11
33.86,6,12
30.37,7,13
35.91,7,14
33.08,8,15
33.45,8,16
```

```
from pyspark.ml.feature import VectorAssembler
import pyspark.sql.types as types

schema = types.StructType() \
    .add("y", types.DoubleType(), True ) \
    .add("x", types.DoubleType(), True ) \
    .add("z", types.DoubleType(), True ) \

data = spark.read.format("csv"). \
        option("header",True).\
        schema(schema). \
        load("twoVariables.csv")

assembler = VectorAssembler(inputCols=["x","z"], outputCol="features")
result = assembler.transform(data)
result.show()
```

```
+-----+---+----+----------+
|    y|  x|   z|  features|
+-----+---+----+----------+
|11.26|1.0| 1.0| [1.0,1.0]|
| 9.67|1.0| 2.0| [1.0,2.0]|
|17.77|2.0| 3.0| [2.0,3.0]|
|11.55|2.0| 4.0| [2.0,4.0]|
|22.18|3.0| 5.0| [3.0,5.0]|
|17.78|3.0| 6.0| [3.0,6.0]|
|19.66|4.0| 7.0| [4.0,7.0]|
| 27.5|4.0| 8.0| [4.0,8.0]|
|28.96|5.0| 9.0| [5.0,9.0]|
|28.53|5.0|10.0|[5.0,10.0]|
|25.45|6.0|11.0|[6.0,11.0]|
|33.86|6.0|12.0|[6.0,12.0]|
|30.37|7.0|13.0|[7.0,13.0]|
|35.91|7.0|14.0|[7.0,14.0]|
|33.08|8.0|15.0|[8.0,15.0]|
|33.45|8.0|16.0|[8.0,16.0]|
+-----+---+----+----------+
```

Fitting the Data

```
import org.apache.spark.ml.regression.LinearRegression
val linearRegression2 = new LinearRegression().setMaxIter(10).setLabelCol(“y")

val lrModel2 = linearRegression2.fit(result)
```

Model

```
Coefficients: [1.0270238095239161,1.1899999999999462] Intercept: 9.449642857142837


RMSE: 3.081205435300251
r2: 0.8658852987773602
```

Actual

y = 2x + z + 5*rand()

RFormula

Taken from R

Describe model via “formula”

$y \sim x + z \longrightarrow y = a + b*x + c*z$

$y \sim x + z - 0 \qquad y = b*x + c*z$

$y \sim x + z + x:z \qquad y = a + b*x + c*z + d*x*z$

dependent
variable

independent
variables

RFormula Example

```
from pyspark.ml.feature import RFormula

data = spark.read.format("csv"). \
        option("header",True).\
        option("inferschema",True).\
        load("linearExactSimple.csv")
linear.show(2)

supervised = RFormula(formula = "y ~ x")

r_formula_model = supervised.fit(data)
prepared_df = model.transform(data)
prepared_df.show()
```

prepared_df is now ready for regression

```
+-----+-----+--------+-----+
|    y|    x|features|label|
+-----+-----+--------+-----+
|  1.0|  1.0|   [1.0]|  1.0|
|  2.0|  2.0|   [2.0]|  2.0|
|  3.0|  3.0|   [3.0]|  3.0|
|  5.0|  5.0|   [5.0]|  5.0|
| 10.0| 10.0|  [10.0]| 10.0|
| 20.0| 20.0|  [20.0]| 20.0|
| 30.0| 30.0|  [30.0]| 30.0|
|100.0|100.0| [100.0]|100.0|
|500.0|500.0| [500.0]|500.0|
+-----+-----+--------+-----+
```

linearExactSimple.csv

```
y,x
1,1
2,2
3,3
5,5
10,10
```

With Two Independent Variables

```
supervised = RFormula(formula = "y ~ x + z")
```

twoVariables.csv

```
y,x,z
11.26,1,1
9.67,1,2
17.77,2,3
11.55,2,4
22.18,3,5
17.78,3,6
19.66,4,7
27.50,4,8
28.96,5,9
28.53,5,10
25.45,6,11
33.86,6,12
30.37,7,13
35.91,7,14
33.08,8,15
33.45,8,16
```

```
+-----+---+----+----------+-----+
|    y|  x|   z|  features|label|
+-----+---+----+----------+-----+
|11.26|1.0| 1.0| [1.0,1.0]|11.26|
| 9.67|1.0| 2.0| [1.0,2.0]| 9.67|
|17.77|2.0| 3.0| [2.0,3.0]|17.77|
|11.55|2.0| 4.0| [2.0,4.0]|11.55|
|22.18|3.0| 5.0| [3.0,5.0]|22.18|
|17.78|3.0| 6.0| [3.0,6.0]|17.78|
|19.66|4.0| 7.0| [4.0,7.0]|19.66|
| 27.5|4.0| 8.0| [4.0,8.0]| 27.5|
|28.96|5.0| 9.0| [5.0,9.0]|28.96|
|28.53|5.0|10.0|[5.0,10.0]|28.53|
|25.45|6.0|11.0|[6.0,11.0]|25.45|
|33.86|6.0|12.0|[6.0,12.0]|33.86|
|30.37|7.0|13.0|[7.0,13.0]|30.37|
|35.91|7.0|14.0|[7.0,14.0]|35.91|
|33.08|8.0|15.0|[8.0,15.0]|33.08|
|33.45|8.0|16.0|[8.0,16.0]|33.45|
+-----+---+----+----------+-----+
```

Hyperparameters

Parameters

- Set before starting the learning process
- Can not be learned from data

Linear Regression hyperparameters

- Feature Column
- Label Column
- Maximum iterations
- Convergence Tolerance
  - When parameters change less than this stop iterating
- Column Weight
  - How much weight to give to each column
- Elastic Net Param

```
class pyspark.ml.regression.LinearRegression(
    featuresCol='features',
    labelCol='label',
    predictionCol='prediction',
    maxIter=100,
    regParam=0.0,
    elasticNetParam=0.0,
    tol=1e-06,
    fitIntercept=True,
    standardization=True,
    solver='auto',
    weightCol=None,
    aggregationDepth=2,
    loss='squaredError',
    epsilon=1.35)
```

loss =

- squaredError
- huber
  - squared error for small errors
  - absolute error for large errors

elasticNetParam =

- 0 -> L2
- 1 -> L1

Elastic Net Parameter

Penalized estimation methods

Reduce or shrink coefficients towards zero

L1 (Ridge Regression)

Shrink many coefficients to zero

Few coefficients with little or no shrinkage

L2 (Lasso)

Tends to produce lots of coefficients close to zero

L2 + L1 (Elastic Net)

How to Select Hyperparmeters?

Knowledge of the data

Experiment with data

- Partition data into three sets
  - Train hyperparmeters
  - Train model
  - Test model

Beware of overfitting!

Grid Search

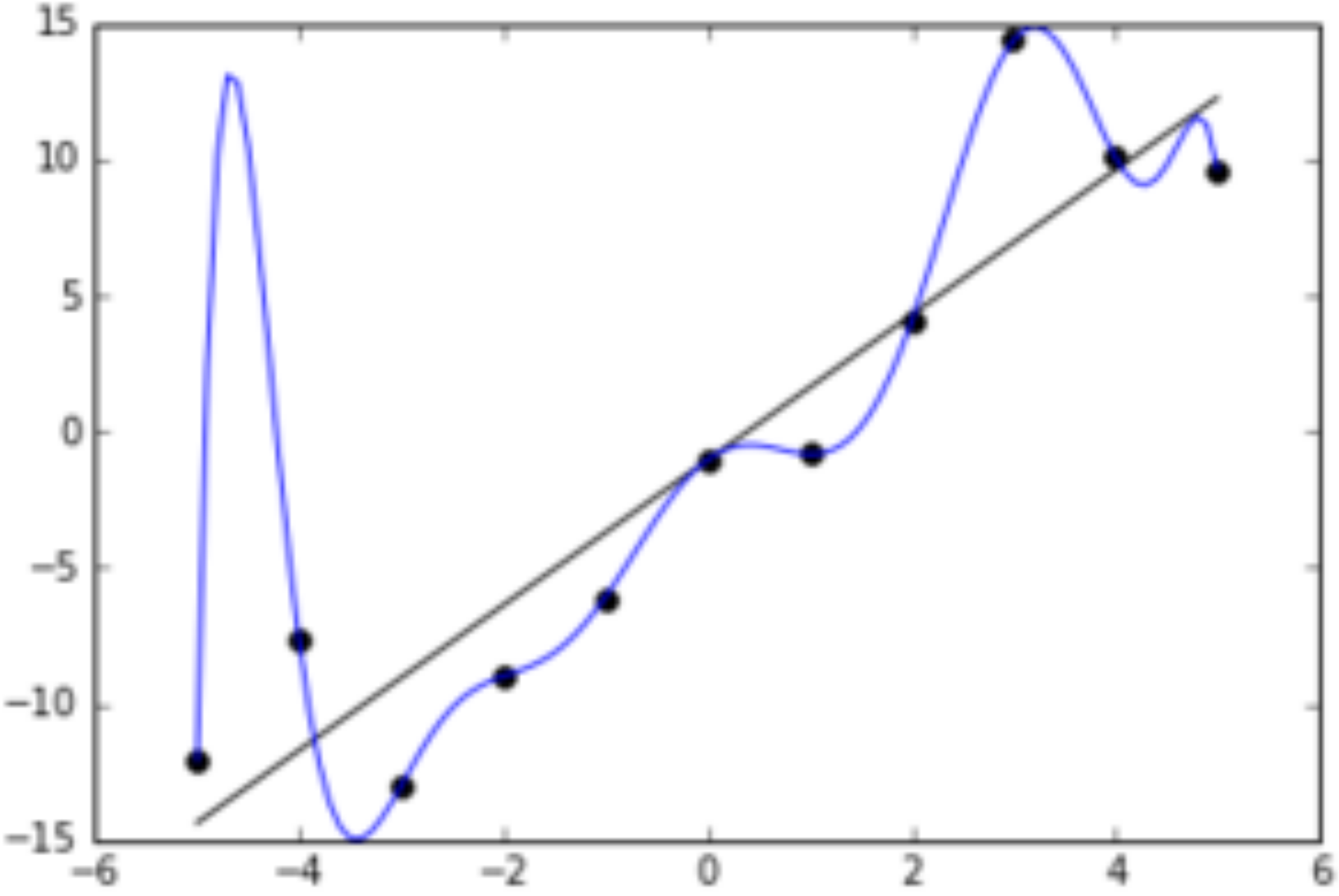
- Select a set of values for each hyperparmeter
- Try all combinations
- Example Later

Overfitting

Model describes random error or noise instead of the underlying relationship

Overfitting occurs when a model is excessively complex,

Too many parameters relative to the number of observations



Generalized Linear Models

Generalized linear regression to handle other cases (distributions)

Linear regression
Logistic regression
Probit regression
Poisson regression
...

Logistic regression
Finite possible outcomes

Categorical Variable

Variable takes on one of limited, usually fixed possible values

- Blood type of a person
- Political party a person will vote for
- State that one lives in

If only two possible values normally encoded as 1 & 0

Categorical variables need to handled differently in regression model

Logistic (Logit) Regression or Logit Model

Regression model where the dependent variable is categorical

Used to predict

If a patient has a disease based on age, sex, blood tests, etc

If a voter will vote Democratic or Republican

If a product will fail

examStudy.csv

Hours Studied & Passing Exam

When only two outcomes encoded 1 & 0

Build model to predict given study time the probability of passing

Pass	Hours
0	0.5
0	0.75
0	1
0	1.25
0	1.5
0	1.75
1	1.75
0	2
1	2.25
0	2.5
1	2.75
0	3
1	3.25
0	3.5
1	4
1	4.25
1	4.5

Logistic Regression Hyperparameters

family:

“multinomial” (multiple labels) or “binary” (two labels).

elasticNetParam:

How to mix L1 and L2 regularization.

fitIntercept:

Boolean, whether or not to fit the intercept.

regParam:

How the inputs should be regularized.

standardization:

Boolean, whether or not to standardize the inputs.

Logistic Regression Training Parameters

maxIter:

Total number of interations before stopping.

tol:

Convergence tolerance for the algorithm.

weightCol:

Name of the weight column to weigh certain rows more than others.

Logistic Regression Prediction Parameters

threshold:

Probability threshold for binary prediction.

Minimum probability for a given class to be predicted.

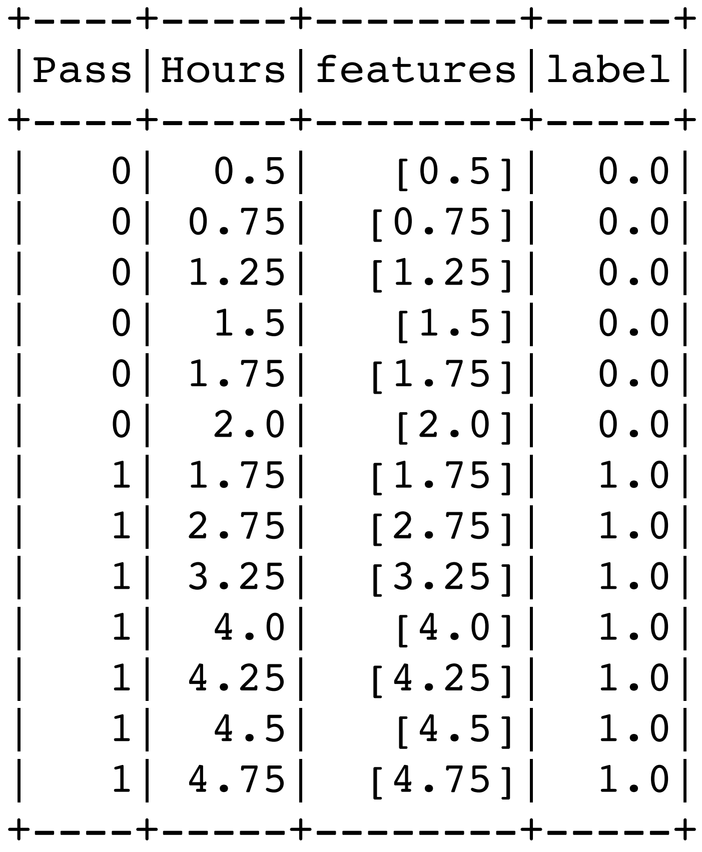
thresholds:

Probability threshold for multinomial prediction.

Minimum probability for a given class to be predicted.

Reading the Data, Applying Formula

```
from pyspark.ml.feature import RFormula
import pyspark.sql.types as types

schema = types.StructType() \
    .add("Pass", types.IntegerType(), True ) \
    .add("Hours", types.DoubleType(), True )

data = spark.read.format("csv"). \
        option("header",True).\
        schema(schema). \
        load("examStudy.csv")

examFormala = RFormula(formula = "Pass ~ Hours")

fitted_rf = examFormala.fit(data)
prepared_df = fitted_rf.transform(data)

(train, test) = prepared_df.randomSplit((0.7, 0.3))
train.show()
```



```
+----+-----+--------+-----+
|Pass|Hours|features|label|
+----+-----+--------+-----+
|   0|  0.5|   [0.5]|  0.0|
|   0| 0.75|  [0.75]|  0.0|
|   0| 1.25|  [1.25]|  0.0|
|   0|  1.5|   [1.5]|  0.0|
|   0| 1.75|  [1.75]|  0.0|
|   0|  2.0|   [2.0]|  0.0|
|   1| 1.75|  [1.75]|  1.0|
|   1| 2.75|  [2.75]|  1.0|
|   1| 3.25|  [3.25]|  1.0|
|   1|  4.0|   [4.0]|  1.0|
|   1| 4.25|  [4.25]|  1.0|
|   1|  4.5|   [4.5]|  1.0|
|   1| 4.75|  [4.75]|  1.0|
+----+-----+--------+-----+
```

Training and Testing Data Sets

Need data to train model

Want to test the model

- Need data but want it to be similar to test data
- Divide data set randomly

```
(train, test) = prepared_df.randomSplit((0.7, 0.3))
```

Fitting the Model

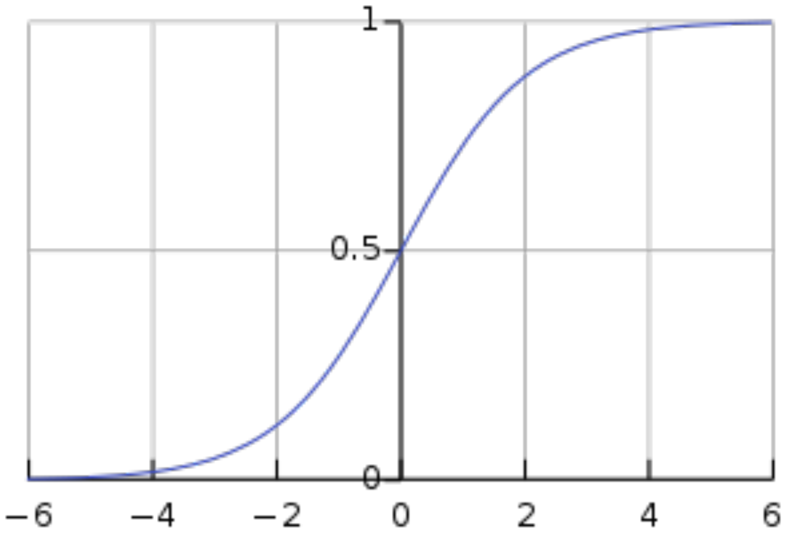
```
from pyspark.ml.classification import LogisticRegression

lr = LogisticRegression()
lr_model = lr.fit(train)
```

Evaluate model using Test data

lr_model.evaluate(test).predictions.show()

```
+----+-----+--------+-----+--------------------+--------------------+----------+
|Pass|Hours|features|label|       rawPrediction|         probability|prediction|
+----+-----+--------+-----+--------------------+--------------------+----------+
|   0|  1.0|   [1.0]|  0.0|[4.27801503632846...|[0.98631958310440...|       0.0|
|   0|  2.5|   [2.5]|  0.0|[-1.8745690184952...|[0.13301393365332...|       1.0|
|   0|  3.0|   [3.0]|  0.0|[-3.9254303701031...|[0.01935176153455...|       1.0|
|   0|  3.5|   [3.5]|  0.0|[-5.9762917217110...|[0.00253179503666...|       1.0|
|   1| 2.25|  [2.25]|  1.0|[-0.8491383426912...|[0.29961364104257...|       1.0|
+----+-----+--------+-----+--------------------+--------------------+----------+
```

Logistic Function



Not fitting data to a line
Fitting it to the logistic function

F(x)= 1/(1 + exp(Intercept + coefficient*x)

```
+----+-----+--------+-----+--------------------+--------------------+----------+
|Pass|Hours|features|label|       rawPrediction|         probability|prediction|
+----+-----+--------+-----+--------------------+--------------------+----------+
|   0|  1.0|   [1.0]|  0.0|[4.27801503632846...|[0.98631958310440...|       0.0|
|   0|  2.5|   [2.5]|  0.0|[-1.8745690184952...|[0.13301393365332...|       1.0|
|   0|  3.0|   [3.0]|  0.0|[-3.9254303701031...|[0.01935176153455...|       1.0|
|   0|  3.5|   [3.5]|  0.0|[-5.9762917217110...|[0.00253179503666...|       1.0|
|   1| 2.25|  [2.25]|  1.0|[-0.8491383426912...|[0.29961364104257...|       1.0|
+----+-----+--------+-----+--------------------+--------------------+----------+
```

Transformers, Estimators, Pipelines

Transformer

- Converts data
- Clean, add features, remove feature, format

Estimators

- Models or variations of same model

Evaluator

- See how an estimator performs

Pipeline

- Specifying transformers and estimators together

Using a Pipeline

Can create a pipeline of transformations and evaluators

Can be applied to multiple data frames

```
from pyspark.ml.classification import LogisticRegression
from pyspark.ml import Pipeline

exam_formula = RFormula(formula = "Pass ~ Hours")
lr = LogisticRegression().setLabelCol('label').setFeaturesCol("features")

pipeline = Pipeline().setStages([exam_formula, lr])
```

Reading data

```
from pyspark.ml.feature import RFormula
import pyspark.sql.types as types

schema = types.StructType() \
    .add("Pass", types.IntegerType(), True ) \
    .add("Hours", types.DoubleType(), True )

data = spark.read.format("csv"). \
        option("header",True).\
        schema(schema). \
        load("examStudy.csv")

(train, test) = data.randomSplit((0.7, 0.3))
```

```
+----+-----+
|Pass|Hours|
+----+-----+
|   0| 0.75|
|   0|  1.0|
|   0| 1.25|
|   0|  1.5|
|   0|  2.0|
|   0|  2.5|
|   0|  3.0|
|   0|  3.5|
|   1| 1.75|
|   1| 2.75|
|   1| 3.25|
|   1|  4.0|
|   1|  4.5|
+----+-----+
```

Using the Pipeline

pipeline_model = pipeline.fit(train)

pipeline_model.transform(test).show()

```
+----+-----+--------+-----+--------------------+--------------------+----------+
|Pass|Hours|features|label|       rawPrediction|         probability|prediction|
+----+-----+--------+-----+--------------------+--------------------+----------+
|   0|  0.5|   [0.5]|  0.0|[3.20443890334995...|[0.96100097970834...|       0.0|
|   0| 1.75|  [1.75]|  0.0|[1.57204004098768...|[0.82807423779535...|       0.0|
|   1| 2.25|  [2.25]|  1.0|[0.91908049604277...|[0.71485471339150...|       0.0|
|   1| 4.25|  [4.25]|  1.0|[-1.6927576837368...|[0.15541352205043...|       1.0|
|   1| 4.75|  [4.75]|  1.0|[-2.3457172286817...|[0.08740679247853...|       1.0|
+----+-----+--------+-----+--------------------+--------------------+----------+
```

Extracting, transforming and selecting features

Feature Extractors

TF-IDF
Word2Vec
CountVectorizer

Feature Selectors

VectorSlicer
RFormula
ChiSqSelector

Feature Transformers

Tokenizer
StopWordsRemover
n-gram
Binarizer
PCA
PolynomialExpansion
Discrete Cosine Transform (DCT)
StringIndexer
IndexToString
OneHotEncoder
VectorIndexer
Interaction
Normalizer
StandardScaler
MinMaxScaler
MaxAbsScaler

Bucketizer
ElementwiseProduct
SQLTransformer
VectorAssembler
QuantileDiscretizer
Imputer

Scaling Data

StandardScaler

transforms a dataset of Vector rows, normalizing each feature to have unit standard deviation and/or zero mean

```
+--------------+--------------------------------+
|      features|scaledFeatures                  |
+--------------+--------------------------------+
|[1.0,0.1,-1.0]| [1.0,0.018156825980064073,-0.5]|
|[2.0,1.1, 1.0]| [2.0,0.19972508578070483,  0.5]|
|[3.0,10.1,3.0]| [3.0,1.8338394239864713,   1.5]|
+--------------+--------------------------------+
```

Scaling Data - MinMaxScaler

transforms a dataset of Vector rows, rescaling each feature to a specific range

```
Features scaled to range: [0.0, 1.0]
+--------------+--------------+
|      features|scaledFeatures|
+--------------+--------------+
|[1.0,0.1,-1.0]| [0.0,0.0,0.0]|
| [2.0,1.1,1.0]| [0.5,0.1,0.5]|
|[3.0,10.1,3.0]| [1.0,1.0,1.0]|
+--------------+--------------+
```

Example

```
from pyspark.ml.feature import MinMaxScaler
from pyspark.ml.linalg import Vectors

dataFrame = spark.createDataFrame((
  (0, Vectors.dense(1.0, 0.1, -1.0)),
  (1, Vectors.dense(2.0, 1.1, 1.0)),
  (2, Vectors.dense(3.0, 10.1, 3.0))
)).toDF("id", "features")

scaler = MinMaxScaler().setInputCol("features").setOutputCol("scaledFeatures")

# Compute summary statistics and generate MinMaxScalerModel
scalerModel = scaler.fit(dataFrame)

# rescale each feature to range [min, max].
scaledData = scalerModel.transform(dataFrame)
scaledData.show()
```

```
+---+--------------+--------------+
| id|      features|scaledFeatures|
+---+--------------+--------------+
|  0|[1.0,0.1,-1.0]| [0.0,0.0,0.0]|
|  1| [2.0,1.1,1.0]| [0.5,0.1,0.5]|
|  2|[3.0,10.1,3.0]| [1.0,1.0,1.0]|
+---+--------------+--------------+
```

Scaling Data MaxAbsScaler

transforms a dataset of Vector rows, rescaling each feature to range [-1, 1]

```
+--------------+----------------+
|      features|  scaledFeatures|
+--------------+----------------+
|[1.0,0.1,-8.0]|[0.25,0.01,-1.0]|
|[2.0,1.0,-4.0]|[0.5, 0.1, -0.5]|
|[4.0,10.0,8.0]|[1.0, 1.0,  1.0]|
+--------------+----------------+
```

Binning Data - Bucketizer

transforms a column of continuous features to a column of feature buckets

splits: Parameter for mapping continuous features into buckets

```
from pyspark.ml.feature import Bucketizer
values = [(0.1,), (0.4,), (1.2,), (1.5,), (float("nan"),), (float("nan"),)]
df = spark.createDataFrame(values, ["values"])
bucketizer = Bucketizer(splits=[-float("inf"), 0.5, 1.4, float("inf")], \
                                    inputCol="values", outputCol="buckets")
bucketed = bucketizer.setHandleInvalid("keep").transform(df).collect()
bucketed
```

```
[Row(values=0.1, buckets=0.0),
 Row(values=0.4, buckets=0.0),
 Row(values=1.2, buckets=1.0),
 Row(values=1.5, buckets=2.0),
 Row(values=nan, buckets=3.0),
 Row(values=nan, buckets=3.0)]
```

Binning Data - Binarizer

Binarization is the process of thresholding numerical features to binary (0/1) features

```
from pyspark.ml.feature import Binarizer

binarizer = Binarizer(). \
  setInputCol("feature"). \
  setOutputCol("binarized_feature"). \
  setThreshold(0.5)
```

```
Binarizer output with Threshold = 0.5
+---+-------+-----------------+
| id|feature|binarized_feature|
+---+-------+-----------------+
|  0|    0.1|              0.0|
|  1|    0.8|              1.0|
|  2|    0.2|              0.0|
+---+-------+-----------------+
```

QuantileDiscretizer

column with continuous features -> a column with binned categorical features

numBuckets = 3

```
 id | hour
----|------
 0  | 18.0
----|------
 1  | 19.0
----|------
 2  | 8.0
----|------
 3  | 5.0
----|------
 4  | 2.2
```

```
 id | hour | result
----|------|------
 0  | 18.0 | 2.0
----|------|------
 1  | 19.0 | 2.0
----|------|------
 2  | 8.0  | 1.0
----|------|------
 3  | 5.0  | 1.0
----|------|------
 4  | 2.2  | 0.0
```

StringIndexer

encodes a string column of labels to a column of label indices

```
 id | category                 id | category | categoryIndex
----|----------             ----|----------|---------------
 0  | a                        0  | a        | 0.0
 1  | b                        1  | b        | 2.0
 2  | c                        2  | c        | 1.0
 3  | a                        3  | a        | 0.0
 4  | a                        4  | a        | 0.0
 5  | c                        5  | c        | 1.0
```

What happens when test data contains category that training data does not?

throw an exception (which is the default)
skip the row containing the unseen label entirely
put unseen labels in a special additional bucket, at index numLabels

OneHotEncoder

maps a column of label indices to a column of binary vectors

allows algorithms which expect continuous features, such as Logistic Regression, to use categorical features

```
+---+--------+-------------+-------------+
| id|category|categoryIndex|  categoryVec|
+---+--------+-------------+-------------+
|  0|       a|          0.0|(2,[0],[1.0])|
|  1|       b|          2.0|    (2,[],[])|
|  2|       c|          1.0|(2,[1],[1.0])|
|  3|       a|          0.0|(2,[0],[1.0])|
|  4|       a|          0.0|(2,[0],[1.0])|
|  5|       c|          1.0|(2,[1],[1.0])|
+---+--------+-------------+-------------+
```

SQLTransformer

implements the transformations which are defined by SQL statement

```
SELECT a, a + b AS a_b FROM __THIS__
SELECT a, SQRT(b) AS b_sqrt FROM __THIS__ where a > 5
SELECT a, b, SUM(c) AS c_sum FROM __THIS__ GROUP BY a, b
```

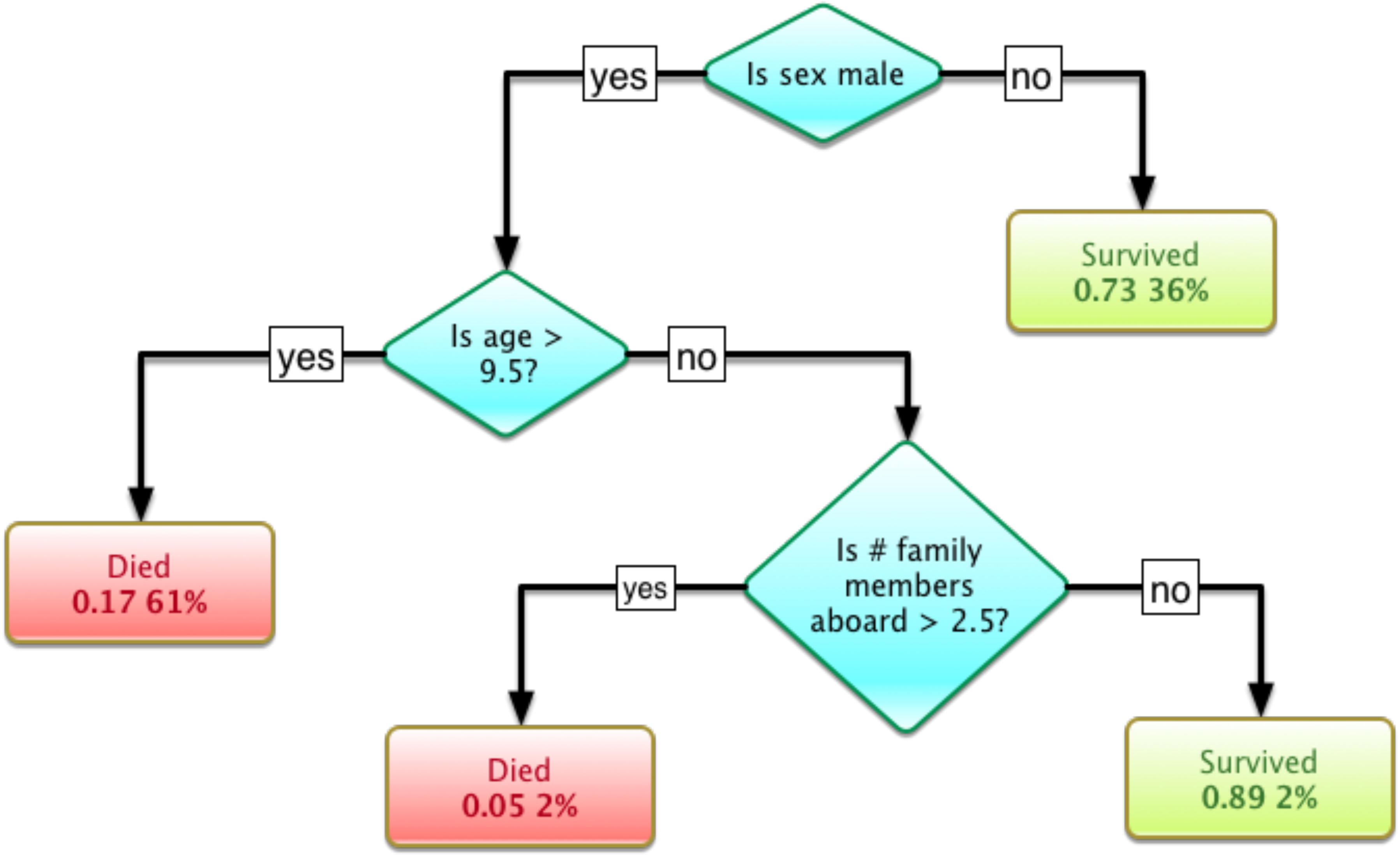
```
from pyspark.ml.feature import SQLTransformer

df = spark.createDataFrame(((0, 1.0, 3.0), (2, 2.0, 5.0))).toDF("id", "v1", "v2")

sqlTrans = SQLTransformer().setStatement(
  "SELECT *, (v1 + v2) AS v3, (v1 * v2) AS v4 FROM __THIS__")

sqlTrans.transform(df).show()
```

```
 id |  v1 |  v2
----|-----|-----
 0  | 1.0 | 3.0
 2  | 2.0 | 5.0
```

```
 id |  v1 |  v2 |  v3 |  v4
----|-----|-----|-----|-----
 0  | 1.0 | 3.0 | 4.0 | 3.0
 2  | 2.0 | 5.0 | 7.0 |10.0
```

Model	Features Count	Training Examples	Output Classes
Logistic Regression	1 to 10 million	no limit	Features x Classes < 10 million
Decision Trees	1,000s	no limit	Features x Classes < 10,000s
Random Forest	10,000s	no limit	Features x Classes < 100,000s
Gradient Boosted Trees	1,000s	no limit	Features x Classes < 10,000s

Decision Trees

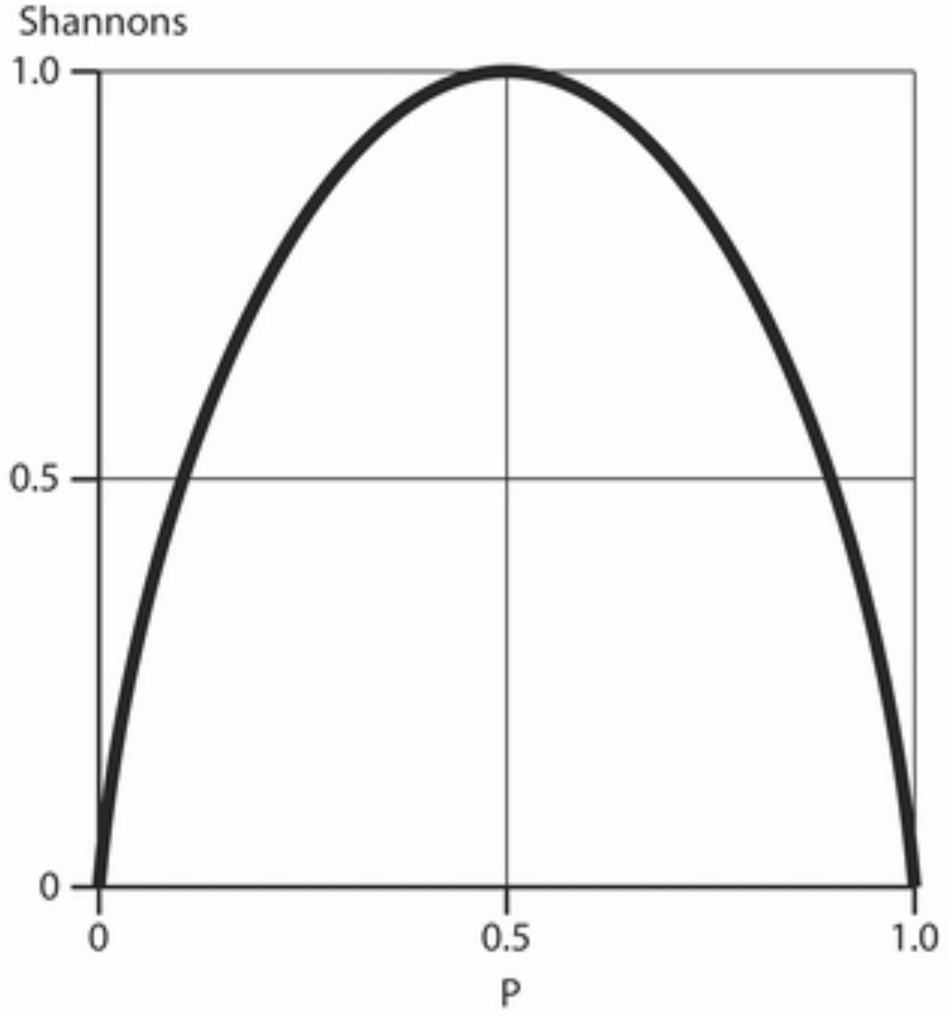


How to Decide Which Question to Ask First

Information

$$I(e) = -log_2 P(e)$$

P(e) = probability of event e



Which card was selected from Deck of Cards

Is it red? (a Heart or a Diamond)
Is it a picture card? (a Jack, Queen, or King)

P(red) = 26/52 = 1/2

I(red) = -log2(1/2) = 1

P(picture) = 12/52 = 3/13

I(picture) = -log2(3/13) = 2.12

P(black) = 26/52 = 1/2

I(black) = -log2(1/2) = 1

P(not picture) = 40/52 = 10/13

I(picture) = -log2(10/13) = 0.38

How to Decide Which Question to Ask First

Entropy - Measure of uncertainty or disorder or order)

$$H(X) = \sum_{i} P(x_i) I(P(x_i))$$

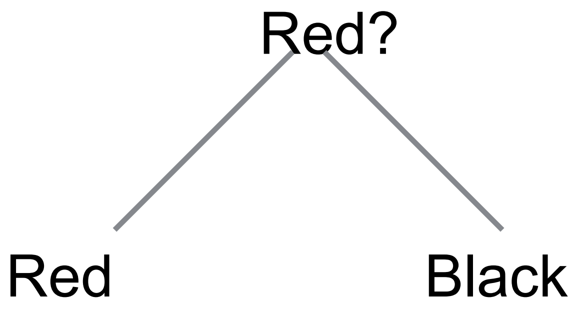
Lower values of H indicate more order, less uncertainty

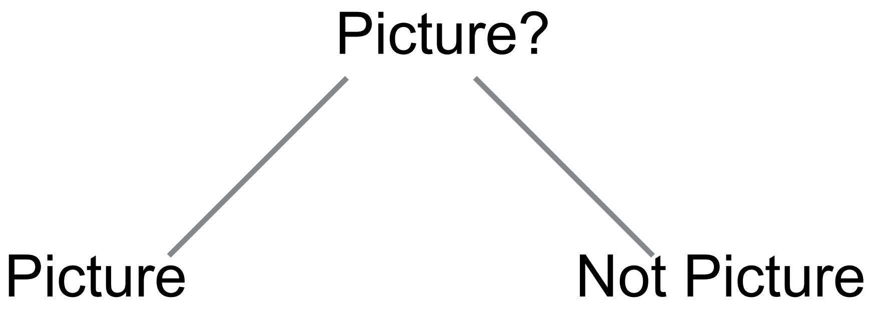
H(red or not red) = P(red)*I(red) + P(not red)*I(not red)
= 1/2 * 1 + 1/2 * 1
= 1

H(picture or not picture) = 3/13 * 2.12 + 10/13 * 0.38
= 0.78

Information Gain

How much entropy decreases





Decision Tree Building

- For root node which feature will produce the most information gain (biggest entropy loss)
- Repeat on each subnode

How to Use

Create a DecisionTreeClassifier

Fit the classifier to your training data

Use the classifier to transform data

But data needs to be in correct format

Need svm format

If categorical the categories need to be integers

Iris Example

```
+------------+-----------+------------+-----------+----------+
|sepal_length|sepal_width|petal_length|petal_width|   species|
+------------+-----------+------------+-----------+----------+
|         5.1|        3.5|         1.4|        0.2|    setosa|
|         4.9|        3.0|         1.4|        0.2|    setosa|
|         4.7|        3.2|         1.3|        0.2|    setosa|
|         4.6|        3.1|         1.5|        0.2|    setosa|

|         7.0|        3.2|         4.7|        1.4|versicolor|
|         6.4|        3.2|         4.5|        1.5|versicolor|
|         6.9|        3.1|         4.9|        1.5|versicolor|
|         5.5|        2.3|         4.0|        1.3|versicolor|
|         6.5|        2.8|         4.6|        1.5|versicolor|
|         5.7|        2.8|         4.5|        1.3|versicolor|

|         6.3|        3.3|         6.0|        2.5| virginica|
|         5.8|        2.7|         5.1|        1.9| virginica|
|         7.1|        3.0|         5.9|        2.1| virginica|
|         6.3|        2.9|         5.6|        1.8| virginica|
```

Reading the File

```
iris = spark.read.format("csv"). \
        option("header",True).\
        option("inferschema",True).\
        load("iris.txt")
iris.schema
```

```
StructType(List(StructField(sepal_length,DoubleType,true),
                StructField(sepal_width,DoubleType,true),
                StructField(petal_length,DoubleType,true),
                StructField(petal_width,DoubleType,true),
                StructField(species,StringType,true)))
```

StringIndexer - convert column to index

```
from pyspark.ml.feature import StringIndexer

irisIndexer = StringIndexer(inputCol="species", outputCol="label").fit(iris)
irisIndexer.transform(iris).show(150)
```

```
+------------+-----------+------------+-----------+----------+------------+
|sepal_length|sepal_width|petal_length|petal_width|   species|label       |
+------------+-----------+------------+-----------+----------+------------+
|         5.1|        3.5|         1.4|        0.2|    setosa|         2.0|
|         4.9|        3.0|         1.4|        0.2|    setosa|         2.0|
|         4.7|        3.2|         1.3|        0.2|    setosa|         2.0|
|         4.6|        3.1|         1.5|        0.2|    setosa|         2.0|

|         7.0|        3.2|         4.7|        1.4|versicolor|         0.0|
|         6.4|        3.2|         4.5|        1.5|versicolor|         0.0|

|         6.0|        3.0|         4.8|        1.8| virginica|         1.0|
|         6.9|        3.1|         5.4|        2.1| virginica|         1.0|
|         6.7|        3.1|         5.6|        2.4| virginica|         1.0|
```

Convert Format

```
from pyspark.ml.feature import VectorAssembler

iris_assembler = VectorAssembler(inputCols=["sepal_length","sepal_width",  \
                                            "petal_length", "petal_width"], outputCol="features")
iris_assembler.transform(iris).show()
```

```
+------------+-----------+------------+-----------+-------+-----------------+
|sepal_length|sepal_width|petal_length|petal_width|species|         features|
+------------+-----------+------------+-----------+-------+-----------------+
|         5.1|        3.5|         1.4|        0.2| setosa|[5.1,3.5,1.4,0.2]|
|         4.9|        3.0|         1.4|        0.2| setosa|[4.9,3.0,1.4,0.2]|
|         4.7|        3.2|         1.3|        0.2| setosa|[4.7,3.2,1.3,0.2]|
|         4.6|        3.1|         1.5|        0.2| setosa|[4.6,3.1,1.5,0.2]|
|         5.0|        3.6|         1.4|        0.2| setosa|[5.0,3.6,1.4,0.2]|
```

Using Pipeline

```
from pyspark.ml import Pipeline

pipeline = Pipeline(stages=[iris_indexer, iris_assembler])
pipeline.fit(iris).transform(iris).show()
```

```
+------------+-----------+------------+-----------+-------+-----+-----------------+
|sepal_length|sepal_width|petal_length|petal_width|species|label|         features|
+------------+-----------+------------+-----------+-------+-----+-----------------+
|         5.1|        3.5|         1.4|        0.2| setosa|  2.0|[5.1,3.5,1.4,0.2]|
|         4.9|        3.0|         1.4|        0.2| setosa|  2.0|[4.9,3.0,1.4,0.2]|
|         4.7|        3.2|         1.3|        0.2| setosa|  2.0|[4.7,3.2,1.3,0.2]|
from pyspark.ml.classification import DecisionTreeClassifier

tree = DecisionTreeClassifier()

pipeline = Pipeline(stages=[iris_indexer, iris_assembler, tree])

(training_data, test_data) = iris.randomSplit([0.7, 0.3])
iris_model = pipeline.fit(training_data)

predictions = iris_model.transform(test_data)
predictions.select("label", "features","probability","prediction").show()
```

```
+-----+-----------------+-------------+----------+
|label|         features|  probability|prediction|
+-----+-----------------+-------------+----------+
|  2.0|[4.3,3.0,1.1,0.1]|[0.0,0.0,1.0]|       2.0|
|  2.0|[4.5,2.3,1.3,0.3]|[0.0,0.0,1.0]|       2.0|
|  2.0|[4.6,3.6,1.0,0.2]|[0.0,0.0,1.0]|       2.0|
|  2.0|[4.8,3.1,1.6,0.2]|[0.0,0.0,1.0]|       2.0|
|  2.0|[4.8,3.4,1.6,0.2]|[0.0,0.0,1.0]|       2.0|
|  0.0|[4.9,2.4,3.3,1.0]|[1.0,0.0,0.0]|       0.0|
|  2.0|[4.9,3.0,1.4,0.2]|[0.0,0.0,1.0]|       2.0|
|  2.0|[4.9,3.1,1.5,0.1]|[0.0,0.0,1.0]|       2.0|
|  0.0|[5.0,2.3,3.3,1.0]|[1.0,0.0,0.0]|       0.0|
|  2.0|[5.0,3.2,1.2,0.2]|[0.0,0.0,1.0]|       2.0|
|  2.0|[5.0,3.4,1.5,0.2]|[0.0,0.0,1.0]|       2.0|
|  2.0|[5.1,3.4,1.5,0.2]|[0.0,0.0,1.0]|       2.0|
|  2.0|[5.1,3.5,1.4,0.3]|[0.0,0.0,1.0]|       2.0|
|  2.0|[5.2,3.4,1.4,0.2]|[0.0,0.0,1.0]|       2.0|
|  0.0|[5.4,3.0,4.5,1.5]|[1.0,0.0,0.0]|       0.0|
|  2.0|[5.4,3.4,1.5,0.4]|[0.0,0.0,1.0]|       2.0|
|  2.0|[5.4,3.4,1.7,0.2]|[0.0,0.0,1.0]|       2.0|
|  0.0|[5.5,2.3,4.0,1.3]|[1.0,0.0,0.0]|       0.0|
|  0.0|[5.7,2.6,3.5,1.0]|[1.0,0.0,0.0]|       0.0|
|  0.0|[5.7,2.8,4.5,1.3]|[1.0,0.0,0.0]|       0.0|
+-----+-----------------+-------------+----------+
```

Some Information about The Tree

Pipeline does not give access to tree (python)

So only use pipeline to transform data

```
pipeline = Pipeline(stages=[iris_indexer, iris_assembler])
pipeline_model = pipeline.fit(training_data)
iris_transformed = pipeline_model.transform(training_data)

iris_tree = DecisionTreeClassifier()
raw_model = iris_tree.fit(iris_transformed)
```

```
raw_model.featureImportances
```

```
SparseVector(4, {0: 0.0291, 2: 0.5744, 3: 0.3965})
```

raw_model.toDebugString

```
DecisionTreeClassificationModel (
  uid=DecisionTreeClassifier_e1ee1b355b47) of depth 4 with 11 nodes
  If (feature 2 <= 2.45)
    Predict: 2.0
  Else (feature 2 > 2.45)
    If (feature 3 <= 1.75)
      If (feature 2 <= 4.95)
        If (feature 0 <= 4.95)
          Predict: 1.0
        Else (feature 0 > 4.95)
          Predict: 0.0
      Else (feature 2 > 4.95)
        If (feature 3 <= 1.65)
          Predict: 1.0
        Else (feature 3 > 1.65)
          Predict: 0.0
    Else (feature 3 > 1.75)
      Predict: 1.0
```

DecisionTree & Iris

Features are continuous

Decision tree bins the values

This dataset bins each feature into two bins

- Feature 2
  - Values <= 2.45
  - Values > 2.45

- Feature 3
  - Depends on values of Feature 2 & 1

Can set maximum number of bins

Can bin values before using tree model

Categorical Features

If have mixed categorical and continuous features

Use VectorIndexer

- Specify how many values needed to be considered continuous

```
from pyspark.ml.feature import VectorIndexer
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer


iris_indexer = StringIndexer(inputCol="species", outputCol="label")
iris_assembler = VectorAssembler(inputCols=["sepal_length","sepal_width", "petal_length",
"petal_width"], outputCol="features_first")
featureIndexer =\
    VectorIndexer(inputCol="features_first", outputCol="features", maxCategories=4)

full_pipeline = Pipeline(stages=[iris_indexer, iris_assembler, featureIndexer])

(training_data, test_data) = iris.randomSplit([0.7, 0.3])
iris_model = full_pipeline.fit(training_data)
iris_transformed = iris_model.transform(training_data)
```

Tree Ensembles

- Gradient-Boosted

- Random Forests
  - Great multiple decision trees

  - Sample training data for each tree

  - Classification
    - Majority vote
  - Regression
    - Average

```
from pyspark.ml.feature import VectorIndexer
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer

iris_indexer = StringIndexer(inputCol="species", outputCol="label")
iris_assembler = VectorAssembler(inputCols=["sepal_length","sepal_width", "petal_length",
"petal_width"], outputCol="features_first")
featureIndexer =\
        VectorIndexer(inputCol="features_first", outputCol="features", maxCategories=4)

full_pipeline = Pipeline(stages=[iris_indexer, iris_assembler, featureIndexer])

(training_data, test_data) = iris.randomSplit([0.7, 0.3])
iris_model = full_pipeline.fit(training_data)
iris_transformed = iris_model.transform(training_data)
test_transformed = iris_model.transform(test_data)
```

```
from pyspark.ml.classification import RandomForestClassifier

forest_classifier = RandomForestClassifier(numTrees = 3)

iris_forest_model = forest_classifier.fit(iris_transformed)
iris_forest_model.trees
```

```
[DecisionTreeClassificationModel (uid=dtc_2eb2f52c851c) of depth 4 with 11 nodes,
 DecisionTreeClassificationModel (uid=dtc_5d145ec5624f) of depth 5 with 11 nodes,
 DecisionTreeClassificationModel (uid=dtc_f9859ed404fb) of depth 4 with 9 nodes]
```

```
prediction = iris_forest_model.transform(test_transformed)
prediction.select("label", "probability","prediction" ).show()
```

```
+-----+-------------+----------+
|label|  probability|prediction|
+-----+-------------+----------+
|  2.0|[0.0,0.0,1.0]|       2.0|
|  2.0|[0.0,0.0,1.0]|       2.0|
|  2.0|[0.0,0.0,1.0]|       2.0|
|  2.0|[0.0,0.0,1.0]|       2.0|
|  2.0|[0.0,0.0,1.0]|       2.0|
|  2.0|[0.0,0.0,1.0]|       2.0|
|  2.0|[0.0,0.0,1.0]|       2.0|
|  0.0|[0.0,1.0,0.0]|       1.0|
|  2.0|[0.0,0.0,1.0]|       2.0|
|  2.0|[0.0,0.0,1.0]|       2.0|
|  2.0|[0.0,0.0,1.0]|       2.0|
```

Decision Trees - Hyperparameters

impurity:

Metric to calculate information gain. “entropy” or “gini”.

maxBins:

Total number of bins used for discretizing continuous features and for choosing how to split on features at each node.

maxDepth:

Determines how deep the total tree can be.

minInfoGain:

Minimum information gain that can be used for a split. A higher value can prevent overfitting.

minInstancePerNode:

Minimum number of instances that need to be in a node. A higher value can prevent overfitting.

Decision Tree

Easy to understand

Easy to interpret

Handles categorical features

Handles multi-class classification

Does not require feature scaling