PySpark ML Features Illustrated

Demonstrations of a few common ML features

Summary

- Feature engineering is the process of using domain knowledge to extract features from raw data via data mining techniques
  - It converts the raw data into a form that can be processed by ML algorithms
- Spark provides many standard methods for transforming raw data into features
  - Examples include: RFormula, StringIndexer, word2vec

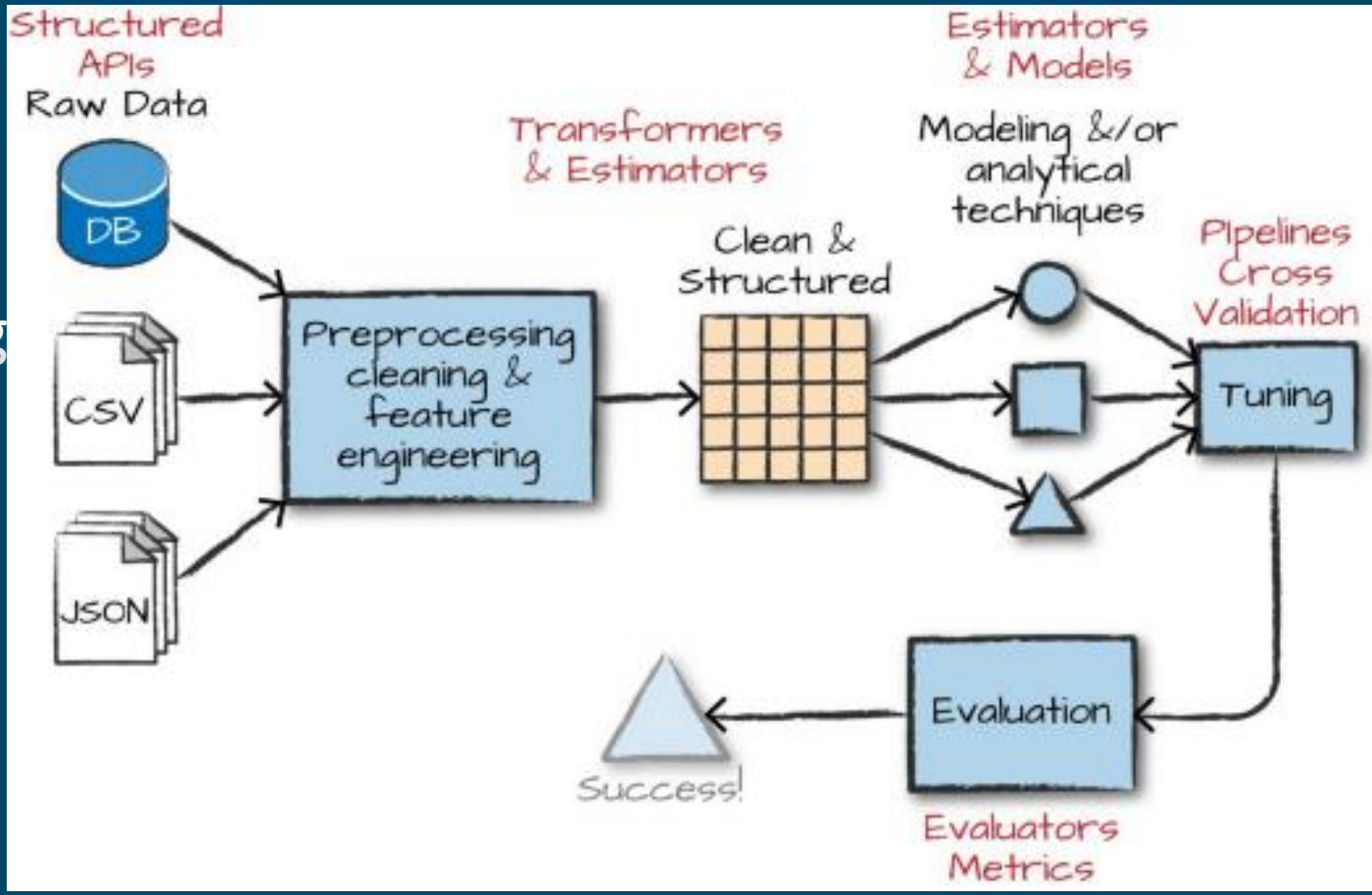
https://en.wikipedia.org/wiki/Feature_engineering

Prepare data for analysis

- Input columns (independent variables, Xs) are converted to a vector of Double and stored in a single Features column
- Predictive column (dependent variable, Y) is stored in a label column (of type Double)
- Examples
  - For recommendation, represent the data as columns of users, items (say movies or books), and ratings
  - For graph analytics, you'll have a DataFrame of vertices and edges

Spark provides ML API's

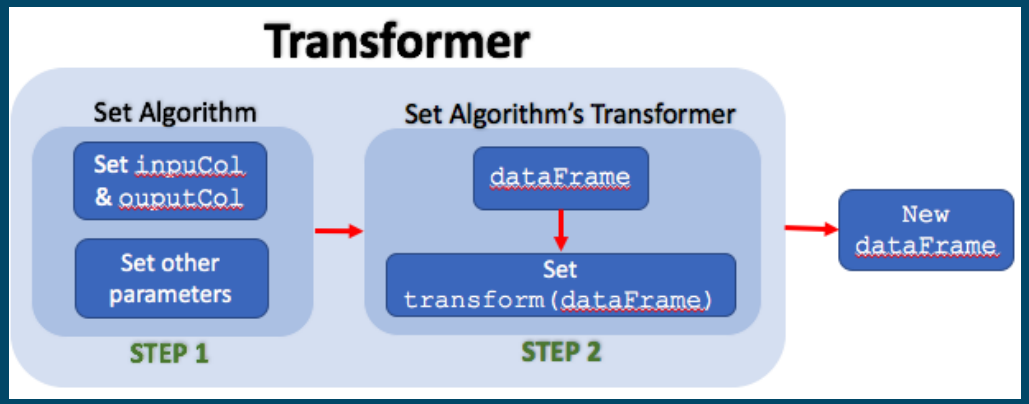
1. Collect data
2. Explore and Visualize data
3. Clean data
4. Transform data for modeling
5. Model data (e.g., regression)
6. Predict using model
7. Evaluate model prediction
8. Visualize results



https://spark.apache.org/docs/latest/ml-pipeline.html

Example transformer: Tokenizer

- Transformer
  - One pass through data to transformation the data




```
1  from pyspark.ml.feature import Tokenizer
2  tkn = Tokenizer().setInputCol("Description").setOutputCol("DescOut")
3  tokenized = tkn.transform(sales.select("Description"))
4  tokenized.show(20, False)
5
```

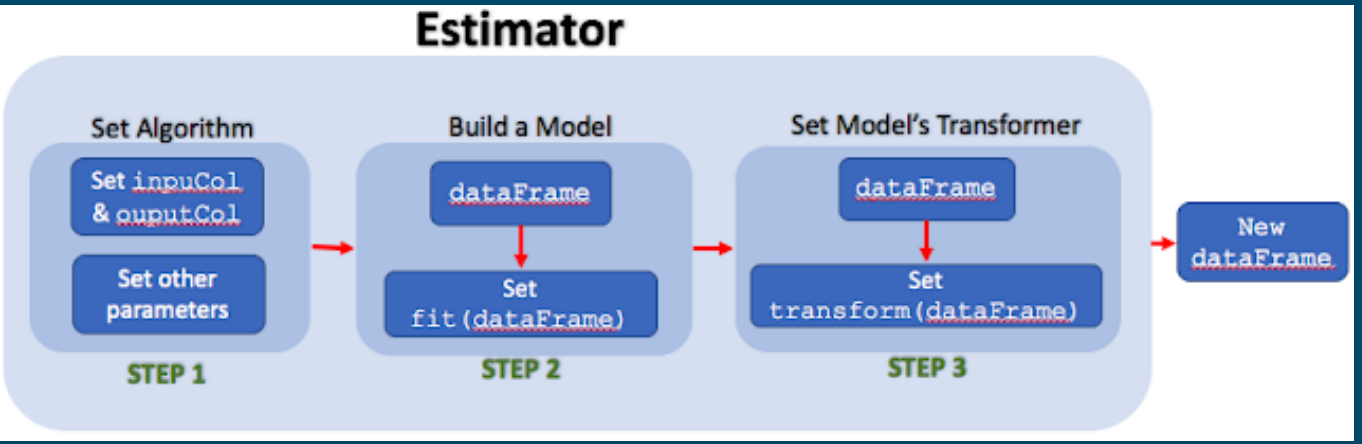
▸ (1) Spark Jobs

▸ tokenized: pyspark.sql.dataframe.DataFrame = [Description: string, DescOut: array]

```
+---------------------------------+-----------------------------------------+
|Description                      |DescOut                                  |
+---------------------------------+-----------------------------------------+
|RABBIT NIGHT LIGHT               |[rabbit, night, light]                   |
|DOUGHNUT LIP GLOSS               |[doughnut, lip, gloss]                   |
|12 MESSAGE CARDS WITH ENVELOPES  |[12, message, cards, with, envelopes]    |
|BLUE HARMONICA IN BOX            |[blue, harmonica, in, box]               |
```

Example estimator: Scaler

- Estimator
  - First pass through data, with fit(), to determine data range for algorithm
    - Scaler returns values, [0,1] with mean of 0 and variance of 1
    - Must review values, fit(), before doing transform()
  - Second pass through data, with transform(), to transform the data



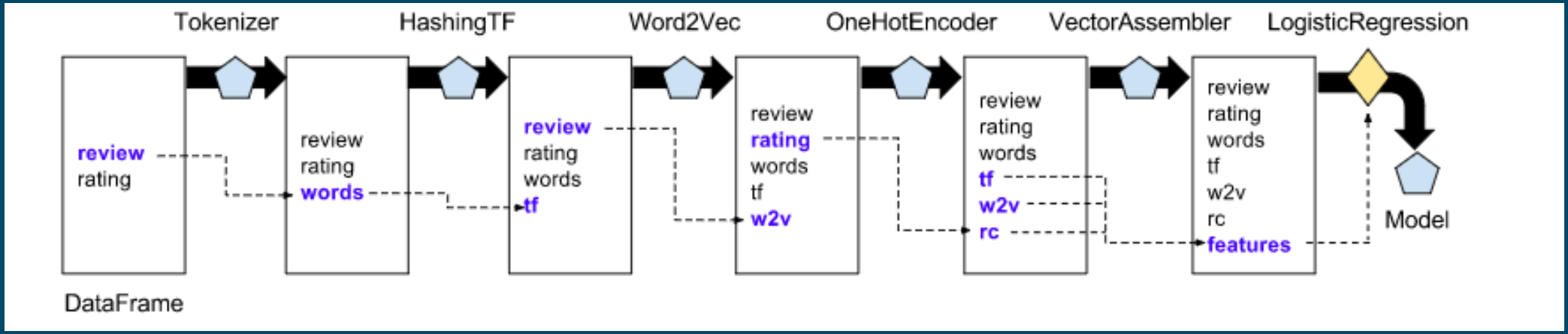

```
1 from pyspark.ml.feature import StandardScaler
2 sScaler = StandardScaler().setInputCol("features")
3 sScaler.fit(scaleDF).transform(scaleDF).show()
4
```

▸ (2) Spark Jobs

```
+---+--------------+------------------------------------+
| id|      features|StandardScaler_6f08b7948339__output|
+---+--------------+------------------------------------+
|  0|[1.0,0.1,-1.0]|                [1.19522860933439...|
|  1| [2.0,1.1,1.0]|                [2.39045721866878...|
|  0|[1.0,0.1,-1.0]|                [1.19522860933439...|
```

Recall ML uses pipeline transformations

- Pipeline
  - Begins with input data
  - Each transformation uses the existing data to add new columns
  - The pipeline ends with columns packaged as the input Features for a model



RFormula creates a features column

- Consider RFormula
  - Data

	Avg_Area_Income	Avg_Area_House_Age	Avg_Area_Number_of_Rooms	Avg_Area_Number_of_Bedrooms	Area_Population	Price	Address
1	79545.45857431678	5.682861321615587	7.009188142792237	4.09	23086.800502686456	1059033.5578701235	208 Michael Ferry Apt. 674
2	79248.64245482568	6.0028998082752425	6.730821019094919	3.09	40173.07217364482	1505890.91484695	188 Johnson Views Suite 079

  - Formula

```
Formula : Price ~ Avg_Area_Income + Avg_Area_House_Age + Avg_Area_Number_of_Rooms + Avg_Area_Number_of_Bedrooms + Area_Population
```

  - features

RFormula creates features & label column using formula (of Xs & &Y)

		Address	features	label
1	55.83597895555	9932 Eric Circles	{"vectorType": "dense", "length": 5, "values": [17796.631189543397, 4.9495570055571125, 6.713905444702088, 2.5, 47162.183643191434]}	302355.83597895555
2	305.577726322	Unit 4700 Box 1880	{"vectorType": "dense", "length": 5, "values": [35454.714659475445, 6.855708363901107, 6.018646502679608, 4.5, 59636.40255302499]}	1077805.577726322

VectorAssembler creates a features column

- Select previously prepared (transformed) columns as input to VectorAssembler, which creates the features

```
columns = df.columns
# Not using Price (label) or address in features
columns.remove('Price')
columns.remove('Address')

# without RFormula
# Select the input columns for the model (and put them into one features column)
#vectorAssembler_features = VectorAssembler(inputCols=columns, outputCol="features")
# Speicfy the regression
#lr = LinearRegression(labelCol ="Price", featuresCol ="features")

# with RFormula
formula = "{} ~ {}".format("Price", " + ".join(columns))
print("Formula : {}".format(formula))
rformula = RFormula(formula = formula)
lr = LinearRegression(labelCol ="label", featuresCol ="features")
# The ML pipeline
#myStages = [rformula, lr]
pipeline = Pipeline(stages=[rformula, lr])
fittedPipe = pipeline.fit(train_data)
```

ml.feature VectorAssembler

- Purpose
  - Combined multiple columns into a single feature vector, of the format (dense vector) required for models
- Examples

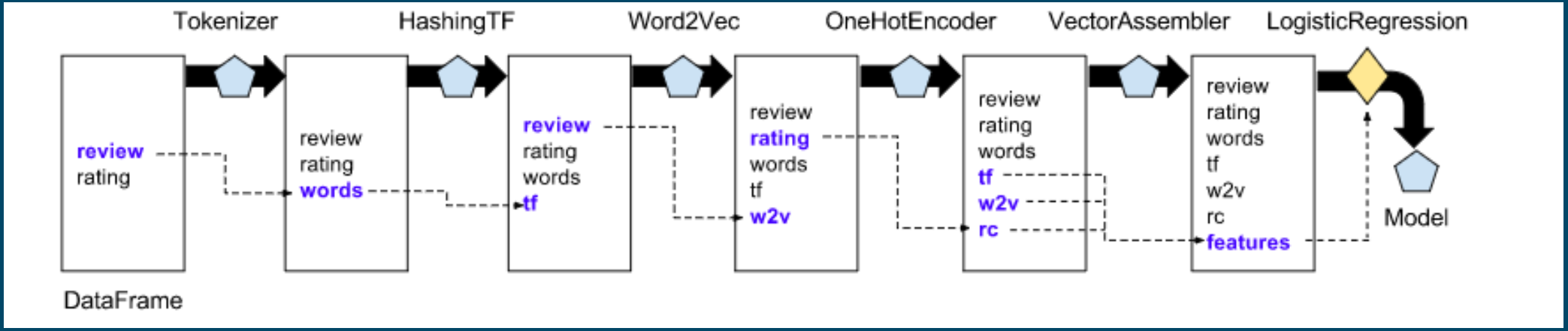
```
from pyspark.ml.feature import VectorAssembler
va = VectorAssembler().setInputCols(["int1", "int2", "int3"])
va.transform(fakeIntDF).show()
```

```
+----+----+----+-----------------------------------------+
|int1|int2|int3|VectorAssembler_403ab93eacd5585ddd2d__output|
+----+----+----+-----------------------------------------+
|   1|   2|   3|                        [1.0,2.0,3.0]|
|   4|   5|   6|                        [4.0,5.0,6.0]|
|   7|   8|   9|                        [7.0,8.0,9.0]|
+----+----+----+-----------------------------------------+
```

```
# Select the input columns for the model (and put them into one features column)
vectorAssembler_features = VectorAssembler(inputCols=["AGE", "MARITAL_STATUS_IX", "PROFESSION_IX"],
outputCol="features")
```

Feature engineering in ML lib

- Select one column as label for ML algorithm
- Use estimators and transformations to create feature columns
- Use VectorAssembler to create the features column



Extracting, transforming and selecting features

This section covers algorithms for working with features, roughly divided into these groups:

- Extraction: Extracting features from “raw” data
- Transformation: Scaling, converting, or modifying features
- Selection: Selecting a subset from a larger set of features
- Locality Sensitive Hashing (LSH): This class of algorithms combines aspects of feature transformation with other algorithms.

Table of Contents

https://spark.apache.org/docs/latest/ml-features

Continuous Features

ml.feature Bucketizer

- Purpose
  - Convert a column of continuous values into discrete ranges
- Parameter for mapping continuous features into buckets.
  - With n+1 splits, there are n buckets.
  - A bucket defined by splits x,y holds values in the range [x,y) except the last bucket, which also includes y.
- Splits array for buckets
  - Minimum value in your splits array must be < the minimum value in your DataFrame.
  - Maximum value in your splits array must be > than the maximum value in your DataFrame.
  - You need to specify at a minimum three values in the splits array, which creates two buckets.

```
# So, defined here, values < 4.5 in bucket1, and bucket2 is values >= 4.5:
splits = [-float("inf"), 4.5, float("inf")]
bucket = Bucketizer(splits=splits, inputCol = "rating", outputCol = "label")
```

ml.feature StandardScaler

- Purpose
  - Convert a data to standard range
- StandardScaler Estimator
  - First, review data (fit), then apply the transformation (transform)
  - Scaler returns values, [0,1] with mean of 0 and variance of 1
  - Must review values, fit(), before doing transform()

```
1 from pyspark.ml.feature import StandardScaler
2 sScaler = StandardScaler().setInputCol("features")
3 sScaler.fit(scaleDF).transform(scaleDF).show()
```

```
+---+--------------+------------------------------------+
| id|      features|StandardScaler_6f08b7948339__output|
+---+--------------+------------------------------------+
|  0|[1.0,0.1,-1.0]|                [1.19522860933439...|
|  1| [2.0,1.1,1.0]|                [2.39045721866878...|
|  0|[1.0,0.1,-1.0]|                [1.19522860933439...|
```

Categorical Features

ml.feature StringIndexer

- Purpose
  - Convert string input (column) to number
  - Finds all unique values, then assigned them numbers
- A label indexer that maps a string column of labels to an ML column of label indices
  - The indices are in [0, numLabels]
  - ordered by label frequencies, with most frequent label index = 0
- Example

```
from pyspark.ml.feature import StringIndexer
lblIndxr = StringIndexer().setInputCol("lab").setOutputCol("labelInd")
idxRes = lblIndxr.fit(simpleDF).transform(simpleDF)
idxRes.show()
```

```
+-----+----+------+------------------+--------+
|color| lab|value1|        value2|labelInd|
+-----+----+------+------------------+--------+
|green|good|     1|14.386294994851129|     1.0|
...
|  red| bad|     2|14.386294994851129|     0.0|
+-----+----+------+------------------+--------+
```

ml.feature IndexToString

- Convert indexed values back to text

```
from pyspark.ml.feature import IndexToString
labelReverse = IndexToString().setInputCol("labelInd")
labelReverse.transform(idxRes).show()
```

```
+-----+----+------+-----------------+--------+-----------------------------+
|color| lab|value1|           value2|labelInd|IndexToString_415...2a0d__output|
+-----+----+------+-----------------+--------+-----------------------------+
|green|good|     1|14.386294994851129|     1.0|                         good|
...
|  red| bad|     2|14.386294994851129|     0.0|                          bad|
+-----+----+------+-----------------+--------+-----------------------------+
```

When to use StringIndexer?

- It converts a String (categorical) variable/column to Double
- Use it when you have a categorical variable in a model
  - It works with models that recognize categorical variable, e.g., Tree models
- Some models don't recognize categorical variables e.g., regressions
  - StringIndexer maps to a range a values, which can be misinterpreted by regression
  - Instead, use StringIndexer and then OneHotEncoder
    - Which maps the various possible values into a vector, where each value is equidistant from each other (i.e., like dummy variables in regression)

One-Hot encoding

- Purpose
  - Maps a categorical feature (string), represented as a label index, to a binary vector with at most a single one-value indicating the presence of a specific feature
  - Ensure that numbers are equal distance apart
    - StringEncoder: some categories have higher numbers than others, even though the categories are not ordered (which is a problem for regression)
    - One-Hot Encoder: converts *n* categories to a binary vector of length *n*, where one value is set to True for each category (aka dummy variables in regression)

```
from pyspark.ml.feature import OneHotEncoder, StringIndexer
lblIndxr = StringIndexer().setInputCol("color").setOutputCol("colorInd")
colorLab = lblIndxr.fit(simpleDF).transform(simpleDF.select("color"))
ohe = OneHotEncoder().setInputCol("colorInd")
ohe.transform(colorLab).show()
```

```
+-----+--------+-------------------------------------+
|color|colorInd|OneHotEncoder_46b5ad1ef147bb355612__output|
+-----+--------+-------------------------------------+
|green|    1.0|                (2,[1],[1.0])|
| blue|    2.0|                    (2,[],[])|
...
|  red|    0.0|                (2,[0],[1.0])|
|  red|    0.0|                (2,[0],[1.0])|
+-----+--------+-------------------------------------+
```

Text Features (for NLP)

ml.feature Tokenizer

- Purpose
  - From input text (sentence), break it into individual terms (words)
- Tokenizer splits on whitespace
- RegexTokenizer create tokens based on regex
- Example

```
from pyspark.ml.feature import Tokenizer
tkn = Tokenizer().setInputCol("Description").setOutputCol("DescOut")
tokenized = tkn.transform(sales.select("Description"))
tokenized.show(20, False)
```

```
+------------------------------------+------------------------------------------+
|Description                         DescOut                                    |
+------------------------------------+------------------------------------------+
|RABBIT NIGHT LIGHT                  |[rabbit, night, light]                    |
|DOUGHNUT LIP GLOSS                  |[doughnut, lip, gloss]                    |
...
|AIRLINE BAG VINTAGE WORLD CHAMPION |[airline, bag, vintage, world,
champion] |
|AIRLINE BAG VINTAGE JET SET BROWN  |[airline, bag, vintage, jet, set,
brown] |
+------------------------------------+------------------------------------------+
```

ml.feature StopWordsRemover

- Purpose
  - Remove unimportant words (non-predictive) from text
  - Common in bag of words processing
- Example

```
stopWrds = StopWordsRemover(inputCol="tokens", outputCol="words",
                            stopWords=StopWordsRemover.loadDefaultStopWords("english"))
```

ml.feature ngram

- Purpose
  - Compute words sequences of length *n*
- NGram takes as input a sequence of strings (e.g. the output of a Tokenizer).
  - The parameter n is used to determine the number of terms in each n-gram.
- Example

```
from pyspark.ml.feature import NGram
unigram = NGram().setInputCol("DescOut").setN(1)
bigram = NGram().setInputCol("DescOut").setN(2)
unigram.transform(tokenized.select("DescOut")).show(False)
bigram.transform(tokenized.select("DescOut")).show(False)
```

ml.feature CountVectorizer

- Purpose
  - Convert a collection of text documents (all rows of a text column) to vectors of token counts.
- CountVectorizer (estimator) can extract a vocabulary
  - The CountVectorizerModel produces sparse representations for the documents over the vocabulary, which can then be passed to other algorithms like LDA
- Example

```
cv = CountVectorizer(inputCol="words", outputCol="features", vocabSize=400)
```

ml.feature HashingTF

- Purpose
  - Compute term frequency vectors from a text column
- Takes sets of terms (BoW) and converts those sets into fixed-length feature vectors.
  - A term is mapped into an index by applying a hash function
  - Term frequencies are calculated based on the mapped indices.
  - To reduce the chance of collision, we can increase the target feature dimension, i.e. the number of buckets of the hash table
- Example

```
hashingTF = HashingTF(inputCol="words", outputCol="rawFeatures", numFeatures=200)
```

ml.feature IDF

- Purpose
  - Compute the Inverse Document Frequency (IDF) given a collection of documents (all rows of a text column).
- The IDFModel takes feature vectors (generally created from HashingTF or CountVectorizer) and scales each feature.
  - Intuitively, it down-weights features which appear frequently in a corpus.
- Example

```
idf1 = IDF(inputCol="words_cfeatures", outputCol="words_ifeatures")
```

Term frequency-inverse document frequency TF-IDF

- TF-IDF is a feature vectorization method widely used in text mining to reflect the importance of a term to a document in the corpus.
- Denote a term by *t*, a document by *d*, and the corpus by *D*.
- Term frequency TF(t,d)
  - the number of times that term t appears in document d,
- Document frequency DF(t,D)
  - the number of documents that contains term t.
- If we only use term frequency to measure the importance, it is very easy to over-emphasize terms that appear very often but carry little information about the document, e.g. “a”, “the”, and “of”.
- Inverse document frequency
  - is a numerical measure of how much information a term provides…

https://spark.apache.org/docs/latest/ml-features.html#tf-idf

TF-IDF

- Inverse document frequency is a numerical measure of how much information a term provides:

$$IDF(t, D) = \log \frac{|D| + 1}{DF(t, D) + 1},$$

- where |D| is the total number of documents in the corpus.
  - logarithm is used
  - if a term appears in all documents, its IDF value becomes 0
- The TF-IDF measure is the product of TF and IDF:
  - TFIDF(t,d,D)=TF(t,d)·IDF(t,D).

*Note that a smoothing term, 1, is applied to avoid dividing by zero for terms outside the corpus

ml.feature TF-IDF

- Apply TF and then IDF to get TF IDF
- Example

```
tokenizer = Tokenizer(inputCol="sentence", outputCol="words")
wordsData = tokenizer.transform(sentenceData)

hashingTF = HashingTF(inputCol="words", outputCol="rawFeatures", numFeatures=20)
featurizedData = hashingTF.transform(wordsData)
# alternatively, CountVectorizer can also be used to get term frequency vectors

idf = IDF(inputCol="rawFeatures", outputCol="features")
idfModel = idf.fit(featurizedData)
rescaledData = idfModel.transform(featurizedData)

rescaledData.select("label", "features").show()
```

https://spark.apache.org/docs/latest/ml-features.html#tf-idf

ml.feature word2vec

- Purpose
  - Compute a numerical vector, in n-space, representing a word
  - Then, average of all words in the document
- word2vec vector can then be used as features for prediction, document similarity calculations
- Example

```
word2Vec = Word2Vec(vectorSize=3, minCount=0, inputCol="text", outputCol="result")
model = word2Vec.fit(documentDF)
```

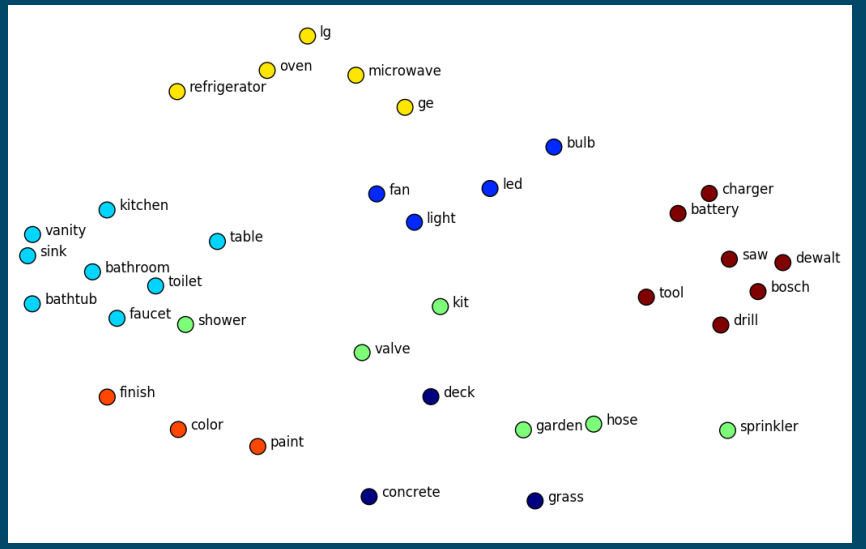
The n in n-space is commonly 300, for the Google word2vec

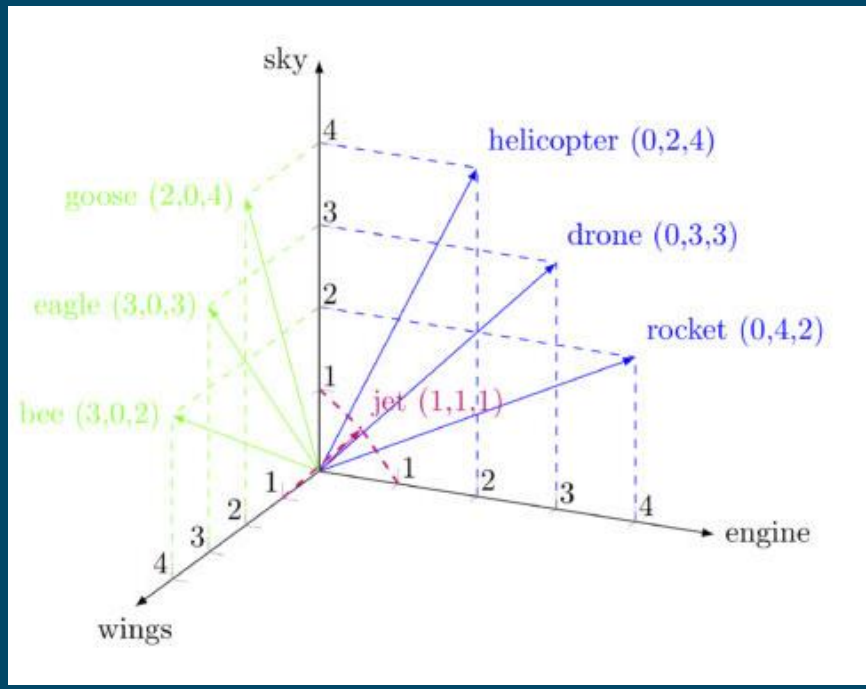
word2vec vector space derived from data

- Words with similar meaning are close in the vector space
  - A word is represented as an array (length = 300) of values
- Using Google's *pre-trained* Word2Vec
  - It's 1.5GB!
  - It includes word vectors for a vocabulary of 3 million words and phrases that they trained on roughly 100 billion words from a Google News dataset.
  - The vector length is 300 features
  - Example
    - word2vect("king") - word2vect("man") + word2vect("woman") = queen

word2vec vector space illustrated

- Vector space
  - [0.x,0.x, ….] length 300
    represents a word in vector space of 300 dimensions
- Examples
  - word2vect(“king”) – word2vect(“man”) + word2vect(“woman”) = queen
  - [walking] - [swimming] + [swam] ~= [walked]
  - [madrid] - [spain] + [france] ~= [paris]



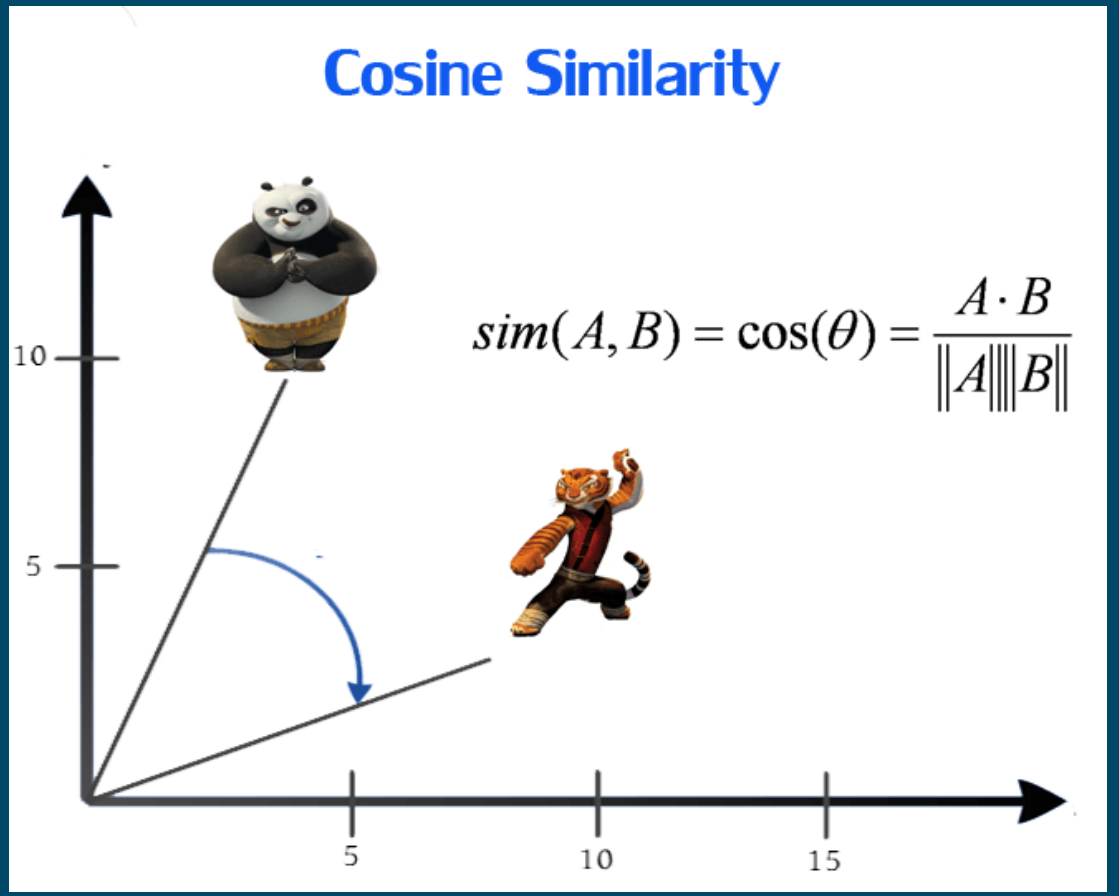


word2vec is a kind of Text encoding

- Word embedding
  - A kind of encoding
  - A vector representation of a word
- Text feature construction
  - Input
    - Word(s)
  - Output
    - Vector (of length n) representing the word(s)

Text similarity

- Once we have a vector for the text (sentence, document), then we can compare them
- Cosine similarity is a measure of similarity between two non-zero vectors of an inner product space that measures the cosine of the angle between them



https://en.wikipedia.org/wiki/Cosine_similarity

Example transformations

3. ML1 Notebook 3

3. Logistic Regression - tokens, buckets (Python)

s6.1 | File | View: Code | Permissions | Run All | Clear | Publish | Comments | Runs

Cmd 1

Adapted from

- https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/8599738367597028/327074921650486/3601578643761083/latest.html
- Update to:
  - Use Databricks dataset for Amazon
- Demonstrates the use of:
  - Bucketizer
  - HashingTF
  - LogisticRegression

Cmd 2

Data Engineering Tasks

A typical data engineer manages the data for fruitful consumption by others, namely data scientists or data anaylists. Besides managing data, they also engage in building complex data pipelines for ETL. One task in ETL is data ingestion and converting raw data into cleansed data. Consider the case here where we import a large Amazon public dataset of product ratings, parse it using one of the Python scripts and then create a permanent table for consumption.

Typical steps:

1. Import raw data from some external source
2. Do some ETL with raw data
3. Preseve cleansed data either as consumable and columnar Parquet files or a SQL table.
4. Create or handle streaming data for ETL

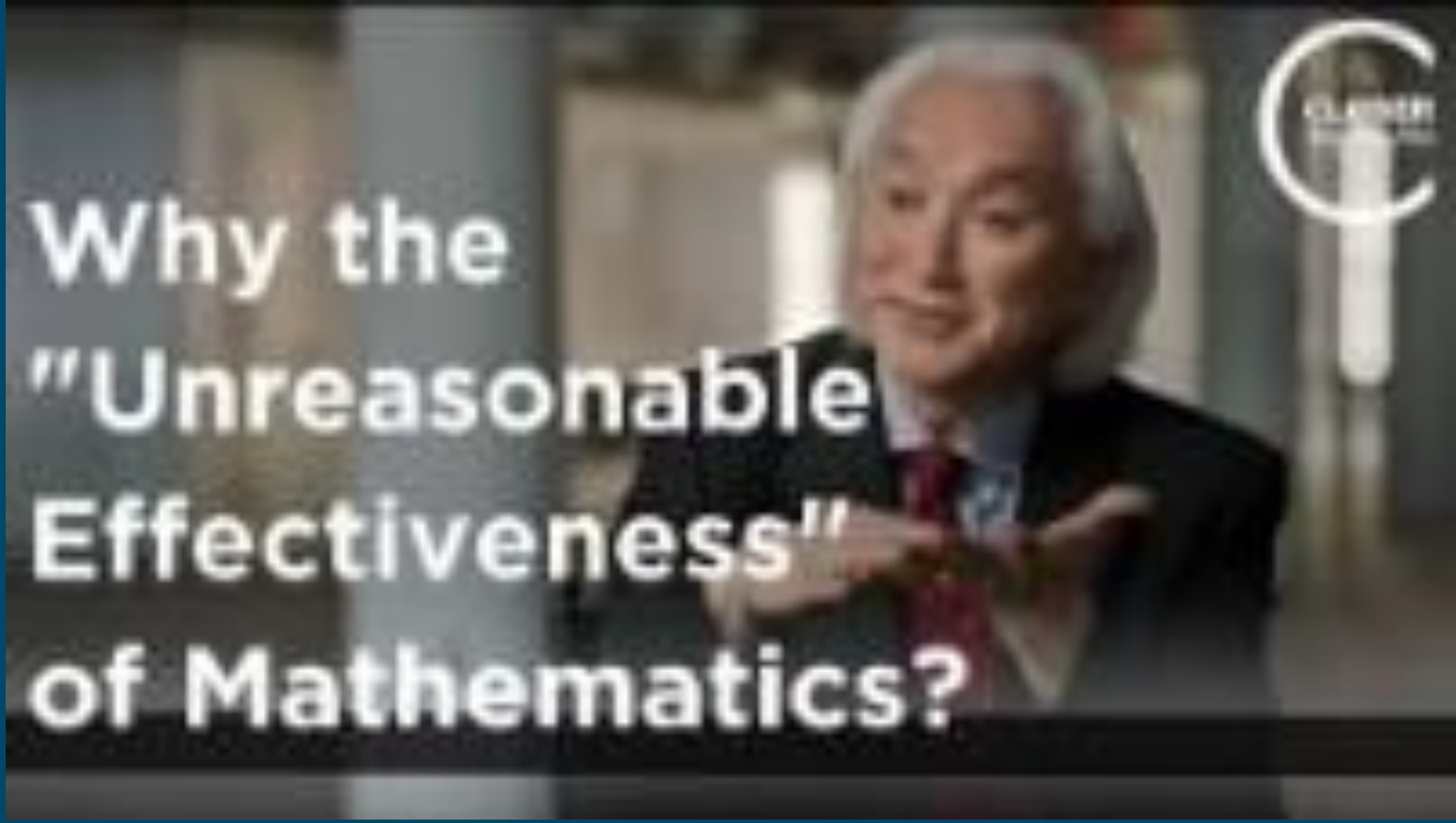
A brief aside about NLP

Math and Nature

- There is a relationship between mathematics and nature
  - mathematical concepts have applicability far beyond the context in which they were originally developed
- Historically
  - Pythagoreans believed that all things were made of numbers
  - Physicist Eugene Wigner, *The Unreasonable Effectiveness of Mathematics* in the Natural Sciences
    - law of gravitation, used to model freely falling bodies on the surface of the earth, this law was extended on the basis "very scanty observations" to describe the motion of the planets, where it "has proved accurate beyond all reasonable expectations".
  - Elegant and powerful explanations
    - $f = ma$ or $e = mc^2$
- In contrast, sciences that involve human beings rather than elementary particles have proven more resistant to elegant mathematics

https://en.wikipedia.org/wiki/The_Unreasonable_Effectiveness_of_Mathematics_in_the_Natural_Sciences


Why the
"Unreasonable
Effectiveness"
of Mathematics?


Why the
"Unreasonable
Effectiveness"
of Mathematics?

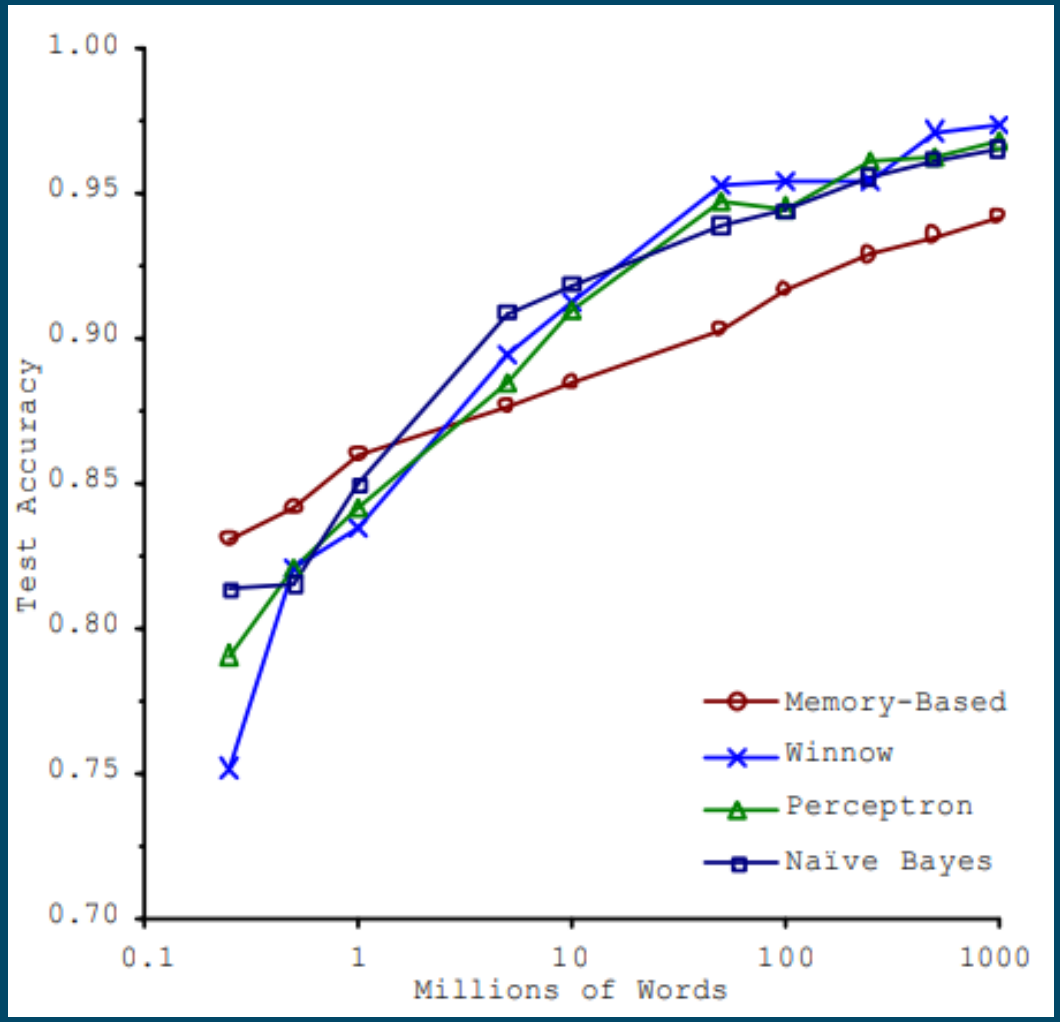
The Unreasonable Effectiveness of Data (Google)

- Very large data sets provide the data from which we can derive a model
  - Very large training sets allowed training, which provided results
    - Statistical speech recognition
      - E.g., closed-caption broadcasts, "OK Google"
    - Statistical machine translation
      - E.g., automated for news agencies in the European Union, Google translate
  - Smaller training sets (with complex algorithms), results in few results
    - document classification, part-of-speech tagging, named-entity recognition, or parsing are not routine tasks, so they have no large corpus available in the wild
- **Invariably, simple models and a lot of data trump more elaborate models based on less data**
  - Cf. success of deep learning

https://research.google.com/pubs/archive/35179.pdf

Algorithm performance depends on data

- Four algorithms to choose the correct use of a word, given a set of words with which it is commonly confused (e.g., principle, principal)
- < 1 million words performance order
  - memory-based learner, naïve Bayes, perceptron, winnow
- 1 billion words performance order
  - winnow, perceptron, naïve Bayes, memory-based learner
  - It's the reverse order
- Algorithm performance depends on data



Banko, M., & Brill, E. (2001, July). Scaling to very very large corpora for natural language disambiguation. In *Proceedings of the 39th annual meeting of the Association for Computational Linguistics* (pp. 26-33).

Large text corpus *is* information

- For many tasks, words and word combinations provide all the representational machinery we need to learn from text
- Three orthogonal NLP problems
  1. choosing a representation language (ngram, deep neural net)
  2. encoding a model in that language (text to model)
  3. performing inference on the model (results from model)

https://research.google.com/pubs/archive/35179.pdf

Spark NLP models

- Encoding the text
  - TD-IDF
  - word2vec
  - ngrams
- Model
  - Regression, LDA, neural net
- Inference
  - Topics, document similarity
- The documents provide the information which our models reveal

ML models in general

- Encode the input
- Model from the encoding
- Inference from the model

Follow the data.

Choose a representation that can use unsupervised learning on unlabeled data, which is so much more plentiful than labeled data.

For natural language applications, trust that human language has already evolved words for the important concepts.
See how far you can go by tying together the words that are already there, rather than by inventing new concepts [features] with clusters of words.

Important to remember

- Feature engineering converts raw data into useful information in a form that can be processed by ML algorithms
- Spark ML lib contains many estimators and transformations for feature engineering
- Spark features column is created from other columns by RFormula or VectorAssembler
- A trend in ML is to use large datasets on generic (simple!?) algorithms (deep learning), trusting that the data inherently contains useful patterns of inference