Large scale text processing pipeline with Spark ML and GraphFrames

Alexey Svyatkovskiy, Kosuke Imai, Jim Pivarski

Princeton University

SPARK SUMMIT EAST 2017

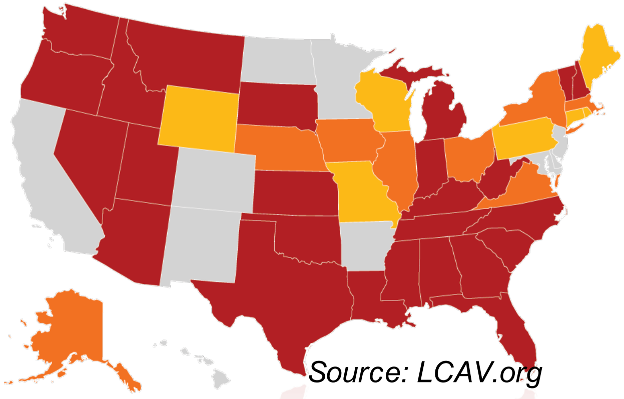
Outline

- Policy diffusion detection in U.S. legislature: the problem
- Solving the problem at scale
  - Apache Spark
  - Text processing pipeline: core modules
- Text processing workflow
  - Data ingestion
  - Pre-processing and feature extraction
  - All-pairs similarity calculation
  - Reformulating problem as a network graph problem
  - Interactive analysis
- All-pairs similarity join
  - Candidate selection: clustering, hashing
- Policy diffusion detection modes
- Interactive analysis with Histogrammar tool
- Conclusion and next steps

Policy diffusion detection: the problem

- Policy diffusion detection is a problem from a wider class of fundamental text mining problems of finding similar items
- Occurs when government decisions in a given jurisdiction are systematically influenced by prior policy choices made in other jurisdictions, in a different state on a different year
- Example: “Stand your ground” bills first introduced in Florida, Michigan and South Carolina 2005
  - A number of states have passed a form of SYG bills in 2012 after T. Martin’s death
- We focus on a type of policy diffusion that can be detected by examining similarity of bill texts

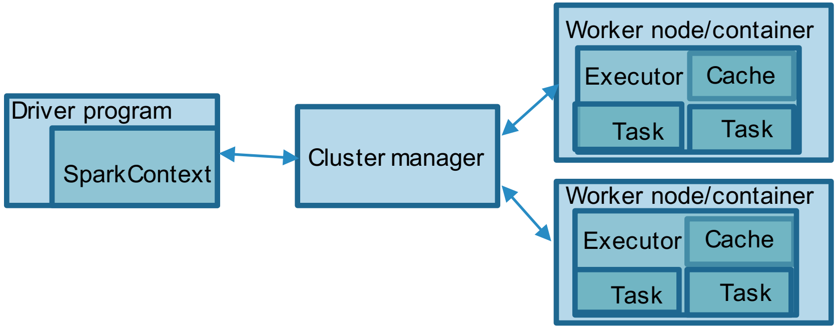
States that have passed SYG laws
States that have passed SYG laws since T. Martin’s death
States that have proposed SYG laws after T. Martin’s death



Source: LCAV.org

Anatomy of Spark applications

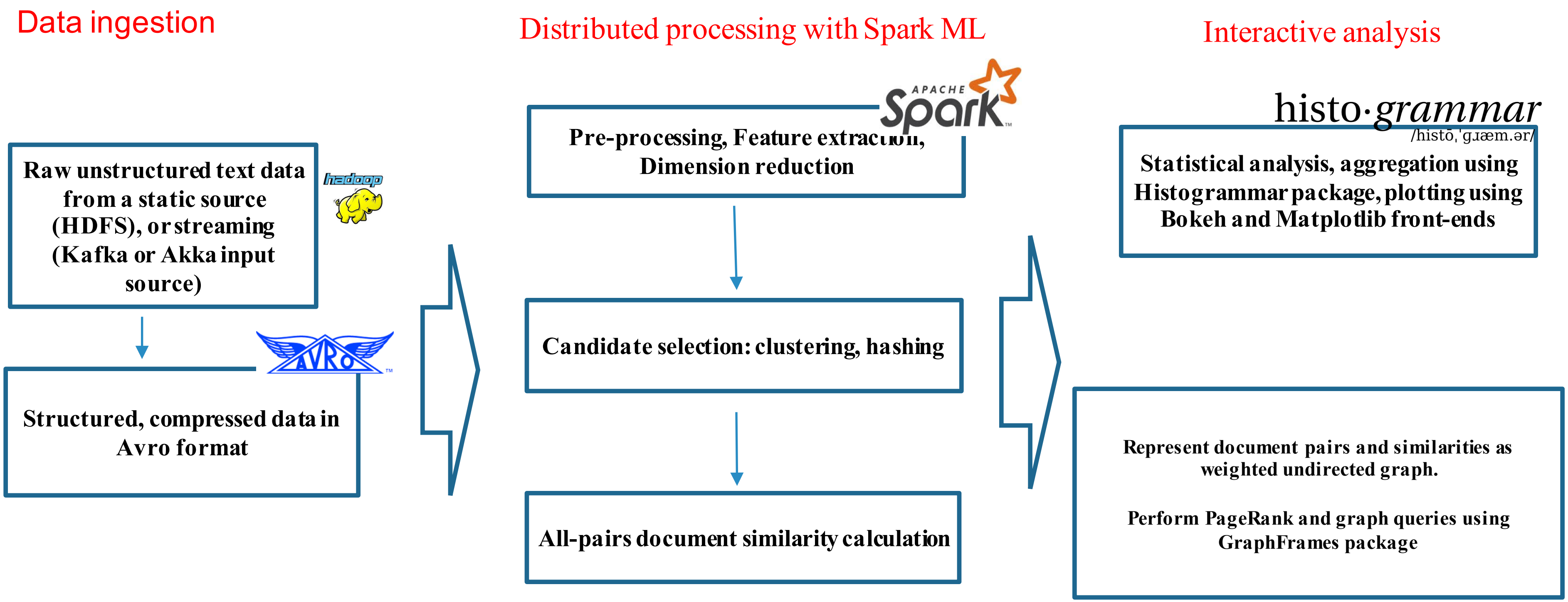
- Spark uses master/worker architecture with central coordinators (drivers) and many distributed workers (executors)
- We choose Scala for our implementation (both driver and executors) because unlike Python and R it is statically typed, and the cost of JVM communication is minimal



Hardware specifications

- A 10 node SGI Linux Hadoop cluster
  - Intel Xeon CPU E5-2680 v2 @ 2.80GHz CPU processors, 256 GB RAM
  - All the servers mounted on one rack and interconnected using a 10 Gigabit Ethernet switch
- Cloudera distribution of Hadoop configured in high-availability mode using two namenodes
  - Schedule Spark applications using YARN
  - Distributed file system (HDFS)
- Alternative configuration uses SLURM resource manager deploying Spark in a standalone mode

Text processing pipeline: core modules



Data ingestion

- Our dataset is based on the LexisNexis StateNet dataset which contains a total of more than 7 million legislative bills from 50 US states from 1991 to 2016
- The initial dataset contains unstructured text documents sub-divided by year and state
- We use Apache Avro serialization framework to store the data and access it efficiently in Spark applications
  - Binary JSON meta-format with schema stored with the data
  - Row-oriented, good for nested schemas. Supports schema evolution

Example raw bill

2005 Bill Text FL S.B. 436

VERSION: Introduced

VERSION-DATE: March 8, 2005

SYNOPSIS: A bill to be entitled

An act relating to the protection of persons and property; creating s. 776.013, F.S.; authorizing a person to use force, including deadly force, against an intruder or attacker in a dwelling, residence, or vehicle under specified circumstances; creating a presumption that a reasonable fear of death or great bodily harm exists under certain circumstances; creating a presumption that a person acts with the intent to use force or violence under specified circumstances; providing definitions; amending ss. 776.012 and 776.031, F.S.; providing that a person is justified in using deadly force under certain

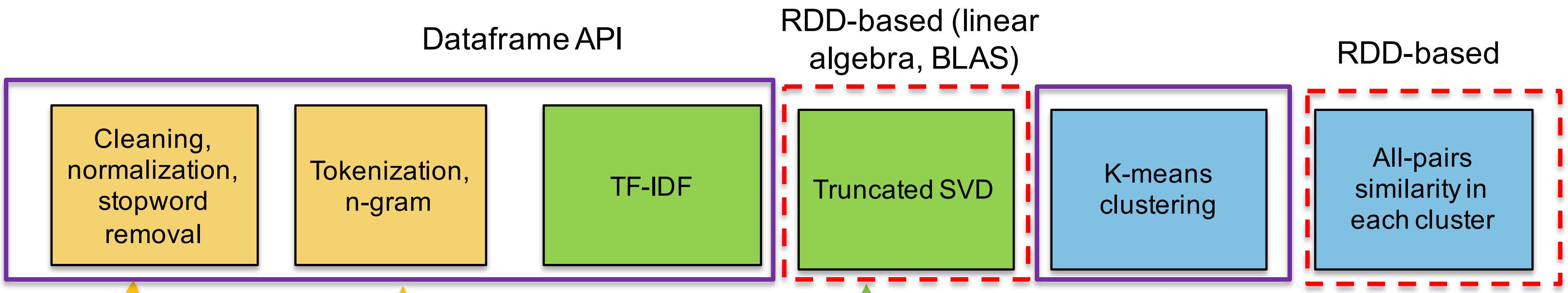
Unique identifier of the bill

Entire contents of the bill as a string, not read into memory during candidate selection and filtering steps, thanks to the Avro schema evolution property

Used to construct predicates and filter the data before calculating joins


```
{"namespace": "bills.avro",
 "type": "record",
 "name": "Bills",
 "fields": [
   {"name": "primary_key", "type": "string"},
   {"name": "content", "type": "string"}
   {"name": "year", "type": "int"},
   {"name": "state", "type": "int"},
   {"name": "docversion", "type": "string"}
 ]
}
```

Analysis workflow1



Raw text of legislative proposals contains spurious white spaces and non-alphanumeric characters, which bare no meaning for analysis and often represent an obstacle for tokenization

Words appearing very frequently across the corpus bare little meaning

Prepare multisets of tokens. N consecutive words, optionally add bag-of-word features.

Feature weighting to reflect importance of a term *t* to a document *d* in corpus *D*

Feature vectors are so high dimension, or so many that they cannot all fit in memory. SVD performs semantic decomposition of TF-IDF matrix and extracts concepts which are most relevant for clustering

Extract candidate documents:
those feature vectors that we need
to test for similarity

Analysis workflow2

Dataframe API

(optional) Cleaning, normalization, stopword removal

N-gram/N-shingle

Min hashing

Locality sensitive hashing

All-pairs similarity in each cluster

Prepare sets of tokens or shingles.
N consecutive words /or characters

Extract minhash signatures:
integer vectors that represent the sets, and reflect their similarity

Extract candidate documents:
a pair of documents that hashes into the same bucket for a fraction of bands makes a candidate pair

Spark ML: a quick review

- Spark ML provides standardized API for machine learning algorithms to make it easier to combine multiple algorithms into a single pipeline or workflow
  - Similarly to scikit-learn Python library

Basic components of a **Pipeline** are:

- **Transformer**: an abstract class to apply a transformation to dataset/dataframes
  - UnaryTransformer abstract class: takes an input column, applies transformation, and output the result as a new column
  - Has a *transform()* method
- **Estimator**: implements an algorithm which can be fit on a dataframe to produce a Transformer. For instance: a learning algorithm is an Estimator which is trained on a dataframe to produce a model.
  - Has a *fit()* method
- **Parameter**: an API to pass parameters to Transformers and Estimators

Putting it all into Pipeline

- **Preprocessing and feature extraction steps**
  - Use *StopWordsRemover, RegexTokenizer, MinHashLSH, HashingTF* transformers
  - *IDF* estimator
- **Prepare custom transformers to plug into Pipeline**
  - Ex: extend *UnaryTransformer,* override *createTransformerFunc* using custom UDF
- **Clustering *KMeans* and *LSH***
- Ability to perform hyper-parameter search and cross-validation

Example custom transformer

```
import org.apache.spark.ml.UnaryTransformer
import org.apache.spark.ml.util.Identifiable
import org.apache.spark.sql.types.{DataType, DataTypes, StringType}

class Cleaner(override val uid: String)
  extends UnaryTransformer[String, String, Cleaner] {

  def this() = this(Identifiable.randomUID("cleaner"))

  def cleanerff(s: String) : String = {
    s.replaceAll("(\\d|,|:|;|\\?|!)", "")
  }

  override protected def createTransformFunc: String => String = {
    cleanerff _
  }

  override protected def validateInputType(inputType: DataType): Unit = {
    require(inputType == StringType)
  }

  override protected def outputDataType: DataType = DataTypes.StringType
}
```

Example ML pipeline (see backup for a full snippet)

```
var idf = new IDF()
     .setInputCol(hashingTF.getOutputCol)
     .setOutputCol("features")

val pipeline = new Pipeline()
     .setStages(Array(cleaner, tokenizer,
             remover, ngram, hashingTF, idf))

 // Fit the pipeline
 val model = pipeline.fit(train)
```

All-pairs similarity: overview

- Our goal is to go beyond identifying the diffusion topics: "stand your ground bills", cyberstalking, marijuana laws. But also to perform an all-pairs comparison
- Previous work in policy diffusion has been unable to make an all-pairs comparison between bills for a lengthy time period because of computational intensity
  - Brute-force all-pairs calculation between the texts of the state bills requires calculating a cross-join, yielding $O(10^{13})$ distinct pairs on the dataset considered
  - As a substitute, scholars studied single topic areas
- Focusing on the document vectors which are likely to be highly similar is essential for all-pairs comparison at scale
- Modern studies employ variations of nearest-neighbor search, locality sensitive hashing (LSH), as well as sampling techniques to select a subset of rows of TF-IDF matrix based on the sparsity (DIMSUM)
- Our approach utilizes clustering and hashing methods (details on the next slide)

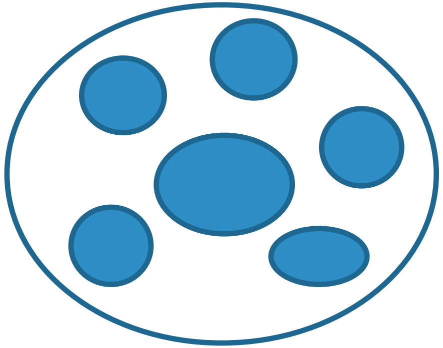
All-pairs similarity, workflow1: clustering

- First approach utilizes K-means clustering to identify groups of candidate documents which are likely to belong to the same diffusion topic, reducing the number of comparisons in the all-pairs similarity join calculation
  - Distance-based clustering using fast square distance calculation in Spark ML
  - Introduce a dataframe column with a cluster label. Perform all-pairs calculation within each cluster

```
def twoSidedJoin(pairs: RDD[(String,String)], hashed_bills: RDD[(String,SparseVector)]):
    RDD[((String,String),(SparseVector,SparseVector))] = {
    val firstjoin = pairs.map({
       case (k1,k2) => (k1, (k1,k2))})
      .join(hashed_bills)
      .map({case (_, ((k1, k2), v1)) => ((k1, k2), v1)})

    val matches = firstjoin.map({
      case ((k1,k2),v1) => (k2, ((k1,k2),v1))})
      .join(hashed_bills)
      .map({case(_, (((k1,k2), v1), v2))=>((k1, k2),(v1, v2))})
    matches
}
```

  - Determine the optimum number of clusters empirically, by repeating the calculation for a range of values of k, scoring on a processing time versus WCSSE plane
  - 150 clusters for a 3 state subset, 400 clusters for the entire dataset
  - 2-3 orders of magnitude less combinatorial pairs to calculate compared to the brute-force approach



Brute force solution is too slow…

All-pairs similarity, workflow2: LSH

- N-shingle features with relatively large N > 5, hashed, converted to sets
- Characteristic matrix instead of TF-IDF matrix (values 0 or 1)
- Extract MinHash signatures for each column (document) using a family of hash functions $h_1(x)$, $h_2(x)$, $h_3(x)$, … $h_n(x)$
  - Hash several times
  - The similarity of two signatures is the fraction of the hash functions in which they agree
- LSH: focus on pairs of signatures which are likely to be from similar documents
  - Hash columns of signature matrix M to many buckets
  - Partition signature matrix into bands of rows. For each band, hash each column into k buckets
  - A pair of documents that hashes into the same bucket for a fraction of bands makes a candidate pair
- Use MinHashLSH class in Spark 2.1

Characteristic matrix:
N-shingles rows
M-documents columns

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

Ex: Family of 4 hash functions

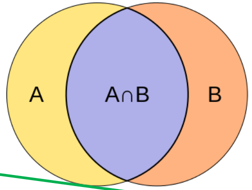
Signature matrix:
K-number of hash functions rows
M-documents columns

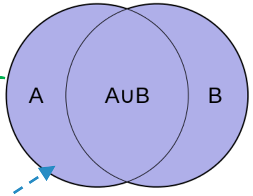
$$\begin{bmatrix} 3 & 1 & 1 & 1 \\ 2 & 1 & 4 & 1 \\ 2 & 2 & 3 & 1 \\ 5 & 2 & 1 & 1 \end{bmatrix}$$

Similarity measures

- The Jaccard similarity between two sets is the size of their intersection divided by the size of their union:

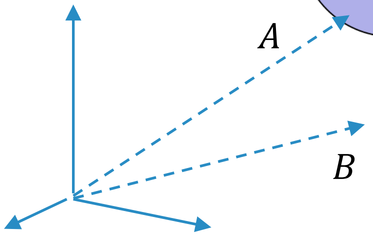
$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$





  - Key distance: consider feature vectors as sets of indices
  - Hash distance: consider dense vector representations
- Cosine distance between feature vectors

$$C(A,B) = \frac{(A \cdot B)}{|A| \cdot |B|}$$



  - Convert distances to similarities assuming inverse proportionality, rescaling it to [0,100] range, adding a regularization term

	Feature type	Similarity measure
Workflow1: K-means clustering	Unigram, TF-IDF, truncated SVD	Cosine, Euclidean
Workflow2: LSH	N-gram with N=5, MinHash	Jaccard

Interactive analysis with Histogrammar tool

- Histogrammar is a suite of composable aggregators with
  - Language independent specification with implementations in Scala and Python
  - Grammar of composable aggregation routines
  - Draws plots in Matplotlib and Bokeh
  - Unlike RDD.histogram, Histogrammar let's one build 2D, profiles, sum-of-squares statistics in an open-ended way

histo·*grammar*
/histō,ˈgɹæm.ər/

http://histogrammar.org

Contact Jim Pivarski or me to contribute!

Example interactive session with Histogrammar

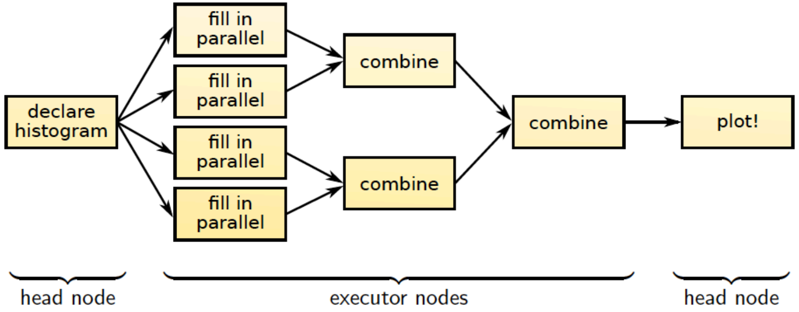
```
import org.dianahep.histogrammar._
import org.dianahep.histogrammar.bokeh._
import org.dianahep.histogrammar.sparksql._
import io.continuum.bokeh._

//UDF to filter data
def stateSelector_udf = udf((pk1: String,pk2: String) =>
{(pk1 contains "FL") || (pk2 contains "FL")})

//load and filter data
val data = spark.read.parquet("path").cache()
val filtered = data.filter(stateSelector_udf(col("pk1"),col("pk2")))

//create and fills the histogram
val hist = filtered.Bin(20, 0, 100, $"similarity")

//plot the histogram and save
val plot = hist.bokeh(glyphType="histogram",glyphSize=3,fillColor=Color.Red)
               .plot(xLabel="Similarity",yLabel="Num. pairs")
save(plot,"cosine_sim.html")
```

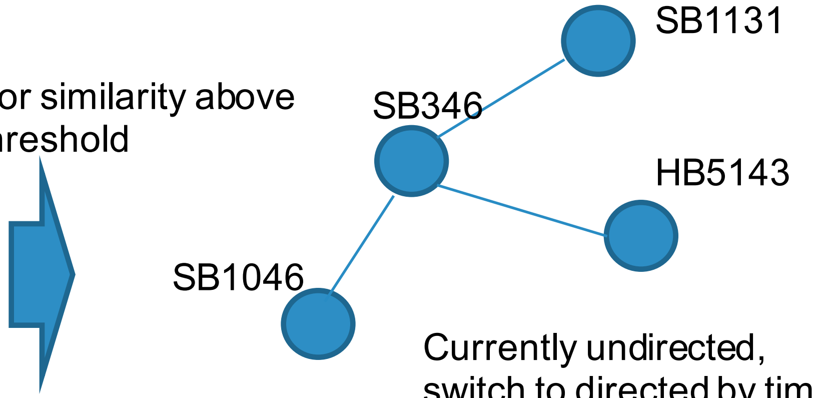


GraphFrames

- GraphFrames is an extension of Spark allowing to perform graph queries and graph algorithms on Spark dataframes
  - A GraphFrame is constructed using two dataframes (a dataframe of nodes and an edge dataframe), allowing to easily integrate the graph processing step into the pipeline along with Spark ML
  - Graph queries: like a Cypher query on a graph database (e.g. Neo4j)
  - Graph algorithms: PageRank, Dijkstra
  - Possibility to eliminate joins

bill1	bill2	similarity
FL/2005/SB436	MI/2005/SB1046	91.38
FL/2005/SB436	MI/2005/HB5143	91.29
FL/2005/SB436	SC/2005/SB1131	82.89

For similarity above threshold



Currently undirected, switch to directed by time

Applications of policy diffusion detection tool

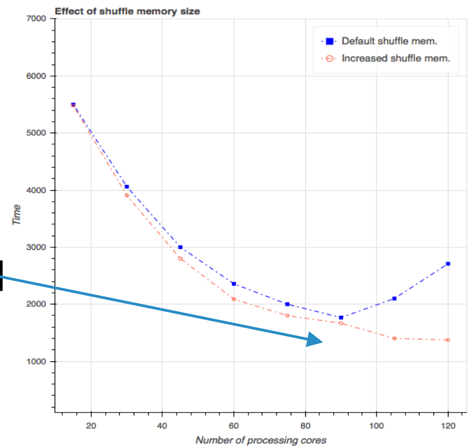
- The policy diffusion detection tool can be used in a number of modes:
  - Identification of groups of diffused bills in the dataset given a diffusion topic (for instance, “Stand your ground” policy, cyberstalking, marijuana laws …)
  - Discovery of diffusion topics: consider top-similarity bills within each cluster, careful filtering of uniform bills and interstate compact bills is necessary as they would show high similarity as well
  - Identification of minimum cost paths connecting two specific legislative proposals on a graph
  - Identification of the most influential US states for policy diffusion
- The political science research paper on applications of the tool is currently in progress

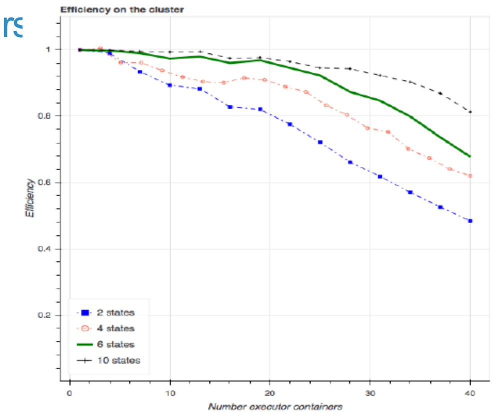
Performance summary, details on the Spark configuration

- The policy diffusion analysis pipeline uses map, filter (narrow), join, and aggregateByKey (wide) transformations
  - Deterministic all-pairs calculation from approach1 involves a two-sided join: heavy shuffle with O(100 TB) intermediate data
  - Benefit from increasing *spark.shuffle.memoryFraction* to 0.6
- Spark applications have been deployed on Hadoop cluster with YARN
  - 40 executor containers, each using 3 executor cores and 15 GB of RAM per JVM
  - Use external shuffle service inside the YARN node manager to improve stability of memory-intensive jobs with larger number of executor containers
  - Custom partitioning to avoid struggler tasks
- Calculate efficiency of parallel execution as

$$E = \frac{T_0}{N_{exec} \cdot T_N}$$

Single executor case





SPARK SUMMIT EAST 2017

Conclusions

- Evaluated Apache Spark framework for the case of data-intensive machine learning problem of policy diffusion detection in US legislature
  - Provided a scalable method to calculate all-pairs similarity based on K-means clustering and MinHashLSH
  - Implemented a text processing pipeline utilizing Apache Avro serialization framework, Spark ML, GraphFrames, and Histogrammar suite of data aggregation primitives
  - Efficiently calculate all-pairs comparison between legislative bills, estimate relationships between bills on a graph, potentially applicable to a wider class of fundamental text mining problems of finding similar items
- Tuned Spark internals (partitioning, shuffle) to obtain good scaling up to O(100) processing cores, yielding 80% parallel efficiency
- Utilized Histogrammar tool as a part of the framework to enable interactive analysis, allows a researcher to perform analysis in Scala language, integrating well with Hadoop ecosystem

Thank You.

Alexey Svyatkovskiy
email: alexeys@princeton.edu
twitter: @asvyatko

Kosuke Imai: kimai@princeton.edu
Jim Pivarski: pivarski@fnal.gov

SPARK SUMMIT EAST 2017

Backup

SSH

Login node

Spark
Hue
YP server
LDAP
…

Service node 1

Zookeeper
Journal node
Primary namenode
httpfs

Service node 2

Zookeeper
Journal node
Resource manager
Hive master

Service node 3

Zookeeper
Journal node
Standby namenode
History server

- A 10 node SGI Linux Hadoop cluster
  - Intel Xeon CPU E5-2680 v2 @ 2.80GHz CPU processors, 256 GB RAM
  - All the servers mounted on one rack and interconnected using a 10 Gigabit Ethernet switch
- Cloudera distribution of Hadoop configured in high-availability mode using two namenodes
  - Schedule Spark applications using YARN
  - Distributed file system (HDFS)

Datanodes

Spark
HDFS
Datanode service

hadoop

TF-IDF

- **TF-IDF weighting**: reflect importance of a term *t* to a document *d* in corpus *D*
  - *HashingTF* transformer, which maps feature to an index by applying MurmurHash 3
  - *IDF* estimator down-weights components which appear frequently in the corpus

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

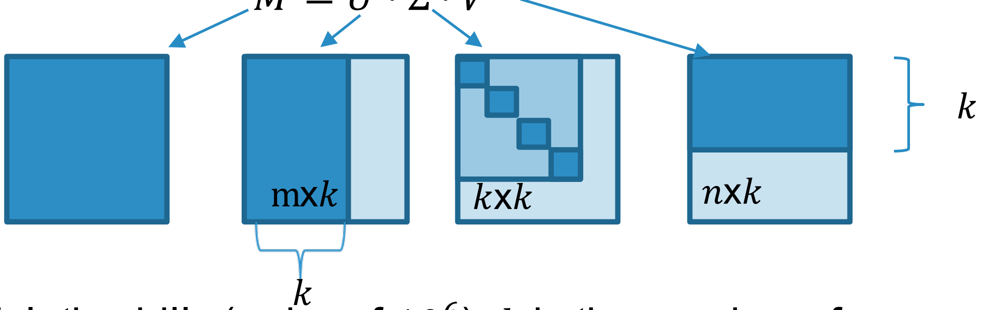
$$TFIDF(t,d,D) = TF(t,d) \bullet IDF(t,D)$$

Dimension reduction: truncated SVD

- SVD is applied to the TF-IDF document-feature matrix to perform semantic decomposition and extract concepts which are most relevant for classification
- RDD-based API, implement RowMatrix transposition, matrix truncation method needed along with SVD
- SVD factorizes the document-feature matrix M $(\mathrm{m}\mathrm{x}n)$ into three matrices: U, Σ, and V, such that:

$$M = U \cdot \Sigma \cdot V^T$$

$\mathrm{m} \gg n \gg k$



Here m is the number of legislative bills (order of 10^6), k is the number of concepts, and n is the number of features (2^{14})

The left singular matrix U is represented as distributed row-matrix, while Σ and V are sufficiently small to fit into Spark driver memory

HistogrammarAggregator class: example

An example of a custom dataset Aggregator class using fill and container from Histogrammar

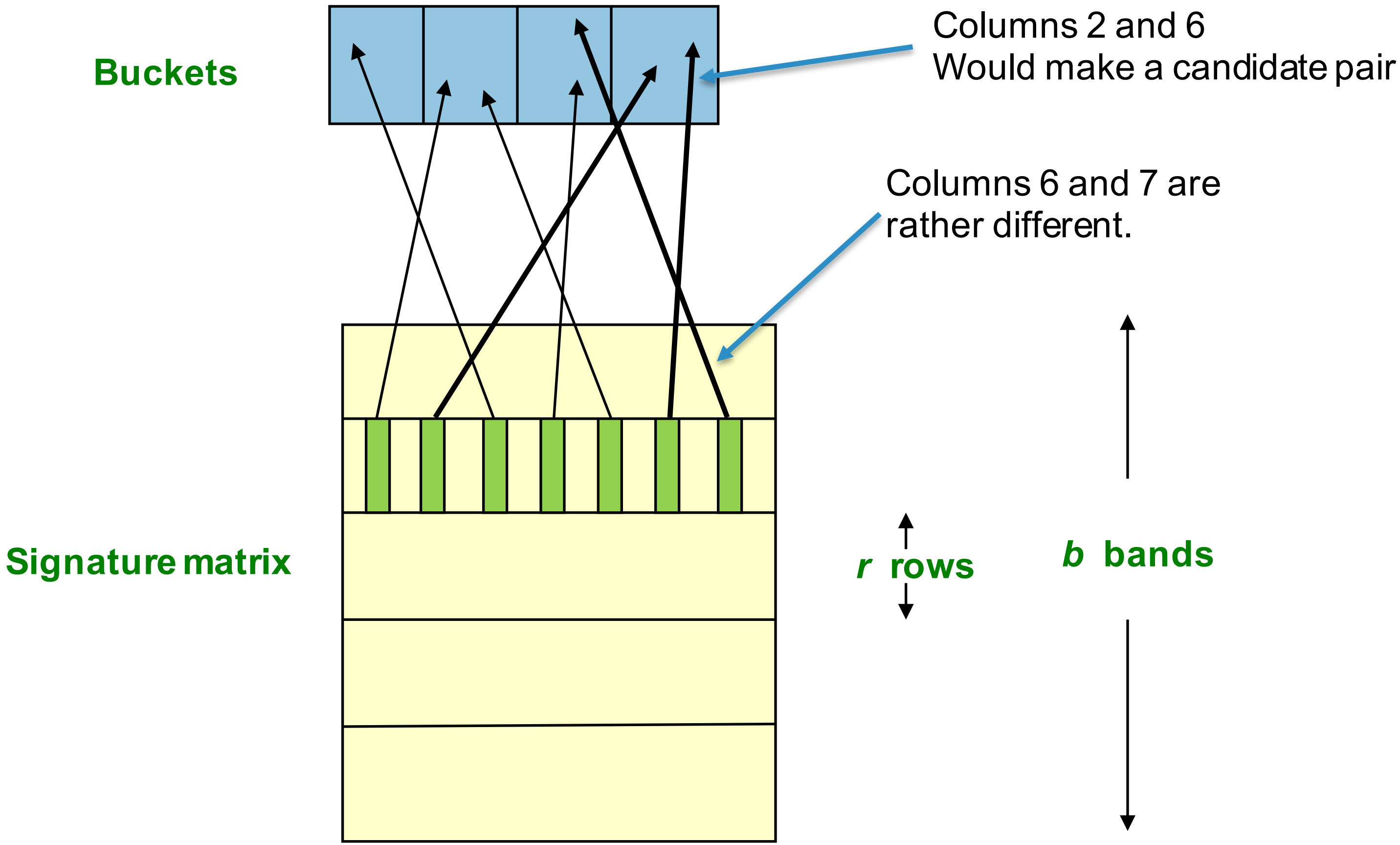
```
package sparksql {
  import org.dianahep.histogrammar.util.Compatible

  class HistogrammarAggregator[CONTAINER <: Container[CONTAINER] with AggregationOnData :
      ClassTag](container: CONTAINER) extends Aggregator[CONTAINER#Datum, CONTAINER, CONTAINER]
      {
    def zero = container
    def reduce(h: CONTAINER, x: CONTAINER#Datum) = {h.fill(x.asInstanceOf[h.Datum]); h}
    def merge(h1: CONTAINER, h2: CONTAINER) = h1 + h2
    def finish(whatever: CONTAINER): CONTAINER = whatever

    override def bufferEncoder: Encoder[CONTAINER] = Encoders.kryo[CONTAINER]
    override def outputEncoder: Encoder[CONTAINER] = Encoders.kryo[CONTAINER]

  }
}
```

Example: Hashing Bands



Example ML pipeline (simplified)

```scala
// Configure an ML pipeline
val cleaner = new Cleaner()
      .setInputCol("content")
      .setOutputCol("cleaned")

val tokenizer = new RegexTokenizer()
      .setInputCol(cleaner.getOutputCol)
      .setOutputCol("words")
      .setPattern("\\W")

val remover = new StopWordsRemover()
      .setInputCol(tokenizer.getOutputCol)
      .setOutputCol("filtered")

val ngram = new NGram()
      .setN(nGramGranularity)
      .setInputCol(remover.getOutputCol)
      .setOutputCol("ngram")
```

```scala
val hashingTF = new HashingTF()
      .setInputCol(ngram.getOutputCol)
      .setOutputCol("keys")
      .setNumFeatures(numTextFeatures)

var idf = new IDF()
      .setInputCol(hashingTF.getOutputCol)
      .setOutputCol("features")

val pipeline = new Pipeline()
      .setStages(Array(cleaner, tokenizer,
            remover, ngram, hashingTF, idf))

// Fit the pipeline
val model = pipeline.fit(train)
```