E6893 Big Data Analytics Lecture 5:

*Big Data Analytics Algorithms — III*

Ching-Yung Lin, Ph.D.
Adjunct Professor, Dept. of Electrical Engineering and Computer Science

October 4th, 2018
Spark ML Classification and Regression

MLlib: Main Guide

- Basic statistics
- Pipelines
- Extracting, transforming and selecting features
- Classification and Regression
- Clustering
- Collaborative filtering
- Frequent Pattern Mining
- Model selection and tuning
- Advanced topics

- Classification
  - Logistic regression
    - Binomial logistic regression
    - Multinomial logistic regression
  - Decision tree classifier
  - Random forest classifier
  - Gradient-boosted tree classifier
  - Multilayer perceptron classifier
  - Linear Support Vector Machine
  - One-vs-Rest classifier (a.k.a. One-vs-All)
  - Naïve Bayes

- Regression
  - Linear regression
  - Generalized linear regression
    - Available families
  - Decision tree regression
  - Random forest regression
  - Gradient-boosted tree regression
  - Survival regression
  - Isotonic regression
Spark ML Classification and Regression

MLlib: Main Guide

- Basic statistics
- Pipelines
- Extracting, transforming and selecting features
- Classification and Regression
- Clustering
- Collaborative filtering
- Frequent Pattern Mining
- Model selection and tuning
- Advanced topics

- Linear methods
- Decision trees
  - Inputs and Outputs
    - Input Columns
    - Output Columns
- Tree Ensembles
  - Random Forests
    - Inputs and Outputs
      - Input Columns
      - Output Columns (Predictions)
  - Gradient-Boosted Trees (GBTs)
    - Inputs and Outputs
      - Input Columns
      - Output Columns (Predictions)
Spark ML Pipeline Example

Pipeline (Estimator)

Raw text → Tokenizer → HashingTF → Feature vectors → Logistic Regression

Pipeline.fit()

Logistic Regression Model
Spark ML Pipeline terms

- **DataFrame**: This ML API uses DataFrame from Spark SQL as an ML dataset, which can hold a variety of data types. E.g., a DataFrame could have different columns storing text, feature vectors, true labels, and predictions.

- **Transformer**: A Transformer is an algorithm which can transform one DataFrame into another DataFrame. E.g., an ML model is a Transformer which transforms a DataFrame with features into a DataFrame with predictions.

- **Estimator**: An Estimator is an algorithm which can be fit on a DataFrame to produce a Transformer. E.g., a learning algorithm is an Estimator which trains on a DataFrame and produces a model.

- **Pipeline**: A Pipeline chains multiple Transformers and Estimators together to specify an ML workflow.

- **Parameter**: All Transformers and Estimators now share a common API for specifying parameters.
Spark ML Pipeline Example

Pipeline (Estimator)

Transformer

Tokenizer → HashingTF → Logistic Regression

Raw text → Words → Feature vectors

Pipeline.fit()

Transformer

Estimator

Logistic Regression Model
```
import org.apache.spark.ml.feature.Tokenizer
val tok = new Tokenizer()

// dataset to transform
val df = Seq(
    (1, "Hello world!"),
    (2, "Here is yet another sentence.")).toDF("id", "sentence")

val tokenized = tok.setInputCol("sentence").setOutputCol("tokens").transform(df)

scala> tokenized.show(truncate = false)
+-----------+------------------+
| id  | sentence          |
+-----------+------------------+
| 1  | Hello world!      |
| 2  | Here is yet another sentence. |
+-----------+------------------+
```

from pyspark.ml.feature import Tokenizer, RegexTokenizer
from pyspark.sql.functions import col, udf
from pyspark.sql.types import IntegerType

sentenceDataFrame = spark.createDataFrame([  
    (0, "Hi I heard about Spark"),  
    (1, "I wish Java could use case classes"),  
    (2, "Logistic, regression, models, are, neat")  
], ["id", "sentence"])

tokenizer = Tokenizer(inputCol="sentence", outputCol="words")

regexTokenizer = RegexTokenizer(inputCol="sentence", outputCol="words", pattern="\\W")  
# alternatively, pattern="\\w", gaps(False)

countTokens = udf(lambda words: len(words), IntegerType())

tokenized = tokenizer.transform(sentenceDataFrame)
tokenized.select("sentence", "words")  
    .withColumn("tokens", countTokens(col("words"))).show(truncate=False)

regexTokenized = regexTokenizer.transform(sentenceDataFrame)
regexTokenized.select("sentence", "words")  
    .withColumn("tokens", countTokens(col("words"))).show(truncate=False)
Vectorization of text

Vector Space Model: Term Frequency (TF)

For example, if the word *horse* is assigned to the 39,905\textsuperscript{th} index of the vector, the word *horse* will correspond to the 39,905\textsuperscript{th} dimension of document vectors. A document’s vectorized form merely consists, then, of the number of times each word occurs in the document, and that value is stored in the vector along that word’s dimension. The dimension of these document vectors can be very large.

Stop Words: \textit{a, an, the, who, what, are, is, was, and so on.}

Stemming:

A stemmer for English, for example, should identify the string "cats" (and possibly "catlike", "catty" etc.) as based on the root "cat", and "stemmer", "stemming", "stemmed" as based on "stem". A stemming algorithm reduces the words "fishing", "fished", and "fisher" to the root word, "fish". On the other hand, "argue", "argued", "argues", "arguing", and "argus" reduce to the stem "argu" (illustrating the case where the stem is not itself a word or root) but "argument" and "arguments" reduce to the stem "argument".
Examples

Assume that we have the following DataFrame with columns id and raw:

<table>
<thead>
<tr>
<th>id</th>
<th>raw</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[I, saw, the, red, baloon]</td>
</tr>
<tr>
<td>1</td>
<td>[Mary, had, a, little, lamb]</td>
</tr>
</tbody>
</table>

Applying `StopWordsRemover` with `raw` as the input column and `filtered` as the output column, we should get the following:

<table>
<thead>
<tr>
<th>id</th>
<th>raw</th>
<th>filtered</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[I, saw, the, red, baloon]</td>
<td>[saw, red, baloon]</td>
</tr>
<tr>
<td>1</td>
<td>[Mary, had, a, little, lamb]</td>
<td>[Mary, little, lamb]</td>
</tr>
</tbody>
</table>

In `filtered`, the stop words “I”, “the”, “had”, and “a” have been filtered out.
from pyspark.ml.feature import StopWordsRemover

sentenceData = spark.createDataFrame([  
    (0, ["I", "saw", "the", "red", "balloon"]),  
    (1, ["Mary", "had", "a", "little", "lamb"])  
], ["id", "raw"])

remover = StopWordsRemover(inputCol="raw", outputCol="filtered")
remover.transform(sentenceData).show(truncate=False)
Most Popular Stemming algorithms

Lookup algorithms

A simple stemmer looks up the inflected form in a lookup table. The advantages of this approach is that it is simple, fast, and easily handles exceptions. The disadvantages are that all inflected forms must be explicitly listed in the table: new or unfamiliar words are not handled, even if they are perfectly regular (e.g. iPads ~ iPad), and the table may be large. For languages with simple morphology, like English, table sizes are modest, but

Suffix-stripping algorithms

Suffix stripping algorithms do not rely on a lookup table that consists of inflected forms and root form relations. Instead, a typically smaller list of "rules" is stored which provides a path for the algorithm, given an input word form, to find its root form. Some examples of the rules include:

- if the word ends in 'ed', remove the 'ed'
- if the word ends in 'ing', remove the 'ing'
- if the word ends in 'ly', remove the 'ly'
n-gram

It was the best of times. it was the worst of times.  

⇒

bigram

It was
was the
the best
best of
of times
of times it
it was
was the
the worst
worst of
of times

Mahout provides a log-likelihood test to reduce the dimensions of n-grams
from pyspark.ml.feature import NGram

wordDataFrame = spark.createDataFrame([  
    (0, ["Hi", "I", "heard", "about", "Spark"]),
    (1, ["I", "wish", "Java", "could", "use", "case", "classes"]),
    (2, ["Logistic", "regression", "models", "are", "neat"])
], ["id", "words"])

ngram = NGram(n=2, inputCol="words", outputCol="ngrams")

ngramDataFrame = ngram.transform(wordDataFrame)
ngramDataFrame.select("ngrams").show(truncate=False)
Word2Vec

The Word2VecModel transforms each document into a vector using the average of all words in the document; this vector can then be used as features for prediction, document similarity calculations, etc.

```python
from pyspark.ml.feature import Word2Vec

# Input data: Each row is a bag of words from a sentence or document.
documentDF = spark.createDataFrame([  
    ("Hi I heard about Spark".split(" "), ),  
    ("I wish Java could use case classes".split(" "), ),  
    ("Logistic regression models are neat".split(" "), )
], ["text"])

# Learn a mapping from words to Vectors.
word2Vec = Word2Vec(vectorSize=3, minCount=0, inputCol="text", outputCol="result")
model = word2Vec.fit(documentDF)

result = model.transform(documentDF)
for row in result.collect():
    text, vector = row
    print("Text: [\%s] => \nVector: \%s\n" % (", ".join(text), str(vector)))
```
CountVectorizer and CountVectorizerModel aim to help convert a collection of text documents to vectors of token counts. When an a-priori dictionary is not available, CountVectorizer can be used as an Estimator to extract the vocabulary, and generates a CountVectorizerModel. The model produces sparse representations for the documents over the vocabulary, which can then be passed to other algorithms like LDA.

Examples

Assume that we have the following DataFrame with columns id and texts:

<table>
<thead>
<tr>
<th>id</th>
<th>texts</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Array(&quot;a&quot;, &quot;b&quot;, &quot;c&quot;)</td>
</tr>
<tr>
<td>1</td>
<td>Array(&quot;a&quot;, &quot;b&quot;, &quot;b&quot;, &quot;c&quot;, &quot;a&quot;)</td>
</tr>
</tbody>
</table>

Each row in texts is a document of type Array[String]. Invoking fit of CountVectorizer produces a CountVectorizerModel with vocabulary (a, b, c). Then the output column “vector” after transformation contains:

<table>
<thead>
<tr>
<th>id</th>
<th>texts</th>
<th>vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Array(&quot;a&quot;, &quot;b&quot;, &quot;c&quot;)</td>
<td>(3, [0,1,2],[1.0,1.0,1.0])</td>
</tr>
<tr>
<td>1</td>
<td>Array(&quot;a&quot;, &quot;b&quot;, &quot;b&quot;, &quot;c&quot;, &quot;a&quot;)</td>
<td>(3, [0,1,2],[2.0,2.0,1.0])</td>
</tr>
</tbody>
</table>

Each vector represents the token counts of the document over the vocabulary.
from pyspark.ml.feature import CountVectorizer

# Input data: Each row is a bag of words with a ID.
df = spark.createDataFrame([  
    (0, "a b c".split(" ")),  
    (1, "a b b c a".split(" "))  
], ["id", "words"])

# fit a CountVectorizerModel from the corpus.
cv = CountVectorizer(inputCol="words", outputCol="features", vocabSize=3, minDF=2.0)

model = cv.fit(df)

result = model.transform(df)
result.show(truncate=False)
FeatureHasher

Feature hashing projects a set of categorical or numerical features into a feature vector of specified dimension (typically substantially smaller than that of the original feature space). This is done using the hashing trick to map features to indices in the feature vector.

The FeatureHasher transformer operates on multiple columns. Each column may contain either numeric or categorical features. Behavior and handling of column data types is as follows:

- **Numeric columns**: For numeric features, the hash value of the column name is used to map the feature value to its index in the feature vector. By default, numeric features are not treated as categorical (even when they are integers). To treat them as categorical, specify the relevant columns using the categoricalCols parameter.

- **String columns**: For categorical features, the hash value of the string “column_name=value” is used to map to the vector index, with an indicator value of 1.0. Thus, categorical features are “one-hot” encoded (similarly to using OneHotEncoder with dropLast=false).

- **Boolean columns**: Boolean values are treated in the same way as string columns. That is, boolean features are represented as “column_name=true” or “column_name=false”, with an indicator value of 1.0.

Null (missing) values are ignored (implicitly zero in the resulting feature vector).
FeatureHasher

Examples

Assume that we have a DataFrame with 4 input columns `real`, `bool`, `stringNum`, and `string`. These different data types as input will illustrate the behavior of the transform to produce a column of feature vectors.

```
real | bool | stringNum | string
---- |-----|----------|------
2.2  | true| 1        | foo  
3.3  | false| 2        | bar  
4.4  | false| 3        | baz  
5.5  | false| 4        | foo  
```

Then the output of `FeatureHasher.transform` on this DataFrame is:

```
real | bool | stringNum | string | features
---- |-----|----------|--------|-----------------------
2.2  | true| 1        | foo    | ((262144, [51871, 63643, 174475, 253195], [1.0, 1.0, 2.2, 1.0]))
3.3  | false| 2        | bar    | ((262144, [6031, 80619, 140467, 174475], [1.0, 1.0, 1.0, 3.3]))
4.4  | false| 3        | baz    | ((262144, [24279, 140467, 174475, 196810], [1.0, 1.0, 4.4, 1.0]))
5.5  | false| 4        | foo    | ((262144, [63643, 140467, 168512, 174475], [1.0, 1.0, 1.0, 5.5]))
```

The resulting feature vectors could then be passed to a learning algorithm.
The value of word is reduced more if it is used frequently across all the documents in the dataset.

To calculate the inverse document frequency, the document frequency (DF) for each word is first calculated. Document frequency is the number of documents the word occurs in. The number of times a word occurs in a document isn’t counted in document frequency. Then, the inverse document frequency or $\text{IDF}_i$ for a word, $w_i$, is

$$\text{IDF}_i = \frac{1}{\text{DF}_i}$$

$$W_i = \text{TF}_i \cdot \text{IDF}_i = \text{TF}_i \cdot \frac{N}{\text{DF}_i}$$

or

$$W_i = \text{TF}_i \cdot \log \frac{N}{\text{DF}_i}$$
TF-IDF example

doc <- c("The sky is blue.", "The sun is bright today.",
         "The sun in the sky is bright.", "We can see the shining sun, the bright sun."
)

<table>
<thead>
<tr>
<th>Term</th>
<th>Docs</th>
</tr>
</thead>
<tbody>
<tr>
<td>blue</td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>bright</td>
<td>0 1 1 1</td>
</tr>
<tr>
<td>can</td>
<td>0 0 0 1</td>
</tr>
<tr>
<td>see</td>
<td>0 0 0 1</td>
</tr>
<tr>
<td>shining</td>
<td>0 0 0 1</td>
</tr>
<tr>
<td>sky</td>
<td>1 0 1 0</td>
</tr>
<tr>
<td>sun</td>
<td>0 1 1 2</td>
</tr>
<tr>
<td>today</td>
<td>0 1 0 0</td>
</tr>
</tbody>
</table>

IDF

<table>
<thead>
<tr>
<th>Term</th>
<th>IDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>blue</td>
<td>0.6931472</td>
</tr>
<tr>
<td>bright</td>
<td>0.0000000</td>
</tr>
<tr>
<td>can</td>
<td>0.6931472</td>
</tr>
<tr>
<td>see</td>
<td>0.6931472</td>
</tr>
<tr>
<td>shining</td>
<td>0.6931472</td>
</tr>
<tr>
<td>sky</td>
<td>0.2876821</td>
</tr>
<tr>
<td>sun</td>
<td>0.0000000</td>
</tr>
<tr>
<td>today</td>
<td>0.6931472</td>
</tr>
</tbody>
</table>
from pyspark.ml.feature import HashingTF, IDF, Tokenizer

sentenceData = spark.createDataFrame(
    [(0.0, "Hi I heard about Spark"),
     (0.0, "I wish Java could use case classes"),
     (1.0, "Logistic regression models are neat")
    ], ["label", "sentence"])

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

df = IDF(inputCol="rawFeatures", outputCol="features")
dfModel = df.fit(featurizedData)
rescaledData = dfModel.transform(featurizedData)
rescaledData.select("label", "features").show()
Examples — using a news corpus

Reuters-21578 dataset: 22 files, each one has 1000 documents except the last one.

http://www.daviddlewis.com/resources/testcollections/reuters21578/

Extraction code:

```
mvn -e -q exec:java
-Dexec.mainClass="org.apache.lucene.benchmark.utils.ExtractReuters"
-Dexec.args="reuters/ reuters-extracted/"
```

Using the extracted folder, run the SequenceFileFromDirectory class. You can use the launcher script from the Mahout root directory to do the same:

```
bin/mahout seqdirectory -c UTF-8
-i examples/reuters-extracted/ -o reuters-seqfiles
```

This will write the Reuters articles in the SequenceFile format. Now the only step left is to convert this data to vectors. To do that, run the SparseVectorsFromSequenceFiles class using the Mahout launcher script:

```
bin/mahout seq2sparse -i reuters-seqfiles/ -o reuters-vectors -ow
```
```python
from pyspark.ml import Pipeline
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.feature import HashingTF, Tokenizer

# Prepare training documents from a list of (id, text, label) tuples.
training = spark.createDataFrame([  
    (0, "a b c d e spark", 1.0),
    (1, "b d", 0.0),
    (2, "spark f g h", 1.0),
    (3, "hadoop mapreduce", 0.0)
], ["id", "text", "label"])

# Configure an ML pipeline, which consists of three stages: tokenizer, hashingTF, and lr.
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.001)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])
```
Spark ML Pipeline Example — classifier

```
PipelineModel
(Transformer)

PipelineModel
.transform()

Tokenizer → HashingTF → Logistic
Regression Model

Raw text → Words → Feature
vectors → Predictions
```
# Prepare test documents, which are unlabeled (id, text) tuples.

test = spark.createDataFrame([  
    (4, "spark i j k"),  
    (5, "l m n"),  
    (6, "spark hadoop spark"),  
    (7, "apache hadoop")  
], ["id", "text"])

# Make predictions on test documents and print columns of interest.

prediction = model.transform(test)
sel ected = prediction.select("id", "text", "probability", "prediction")

for row in selected.collect():
    rid, text, prob, prediction = row

    print("(%d, %s) --> prob=%s, prediction=%r" % (rid, text, str(prob), prediction))
Spark Clustering

- K-means
  - Input Columns
  - Output Columns
- Latent Dirichlet allocation (LDA)
- Bisecting k-means
- Gaussian Mixture Model (GMM)
  - Input Columns
  - Output Columns
Clustering — Gaussian Mixture Models

(Left) Fit with one Gaussian distribution (Right) Fit with Gaussian mixture model with two components
Gaussian Mixture Model

One-dimensional Model

\[ p(x) = \sum_{i=1}^{K} \phi_i N(x \mid \mu_i, \sigma_i) \]

\[ N(x \mid \mu_i, \sigma_i) = \frac{1}{\sigma_i \sqrt{2\pi}} \exp \left( -\frac{(x - \mu_i)^2}{2\sigma_i^2} \right) \]

\[ \sum_{i=1}^{K} \phi_i = 1 \]
Gaussian Mixture Model

Multi-dimensional Model

\[ p(\mathbf{x}) = \sum_{i=1}^{K} \phi_i \mathcal{N}(\mathbf{x} | \bar{\mu}_i, \Sigma_i) \]

\[ \mathcal{N}(\mathbf{x} | \bar{\mu}_i, \Sigma_i) = \frac{1}{\sqrt{(2\pi)^K |\Sigma_i|}} \exp \left( -\frac{1}{2} (\mathbf{x} - \bar{\mu}_i)^T \Sigma_i^{-1} (\mathbf{x} - \bar{\mu}_i) \right) \]

\[ \sum_{i=1}^{K} \phi_i = 1 \]
Gaussian Mixture Model — EM

Expectation maximization (EM) is a numerical technique for maximum likelihood estimation, and is usually used when closed form expressions for updating the model parameters can be calculated (which will be shown below). Expectation maximization is an iterative algorithm and has the convenient property that the maximum likelihood of the data strictly increases with each subsequent iteration, meaning it is guaranteed to approach a local maximum or saddle point.

https://brilliant.org/wiki/gaussian-mixture-model/
Gaussian Mixture Model spark code

### Input Columns

<table>
<thead>
<tr>
<th>Param name</th>
<th>Type(s)</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>featuresCol</td>
<td>Vector</td>
<td>&quot;features&quot;</td>
<td>Feature vector</td>
</tr>
</tbody>
</table>

### Output Columns

<table>
<thead>
<tr>
<th>Param name</th>
<th>Type(s)</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>predictionCol</td>
<td>Int</td>
<td>&quot;prediction&quot;</td>
<td>Predicted cluster center</td>
</tr>
<tr>
<td>probabilityCol</td>
<td>Vector</td>
<td>&quot;probability&quot;</td>
<td>Probability of each cluster</td>
</tr>
</tbody>
</table>

```python
from pyspark.ml.clustering import GaussianMixture

# loads data
dataset = spark.read.format("libsvm").load("data/mllib/sample_kmeans_data.txt")

gmm = GaussianMixture().setK(2).setSeed(538009335)
model = gmm.fit(dataset)

print("Gaussians shown as a DataFrame: ")
model.gaussiansDF.show(truncate=False)
```
Goal – categorize the documents into topics

- Each document is a probability distribution over topics
- Each topic is a probability distribution over words

The probability of $i$th word in a given document

$$P(w_i) = \sum_{j=1}^{T} P(w_i | z_i = j) P(z_i = j)$$

- The probability of the word $w_i$ under the $j$th topic $\theta_j^{(i)}$
- The probability of choosing a word from the $j$th topic in the current document $\phi_w^{(j)}$
LDA (cont.)

INPUT:
- document-word counts
  - $D$ documents, $W$ words

OUTPUT:
- likely topics for a document
  \[ P(z|w) \propto P(w|z)P(z) \]

- Parameters can be estimated by Gibbs Sampling
- Outperform Latent Semantic Analysis (LSA) and Probabilistic LSA in various experiments [Blei et al. 2003]

Bayesian approach: use priors
- Mixture weights $\sim$ Dirichlet($\alpha$)
- Mixture components $\sim$ Dirichlet($\beta$)
Traditional Content Clustering

**Clustering:**
Partition the feature space into segments based on training documents. Each segment represents a topic / category. (_topic detection)

**Hard clustering:** e.g., K-mean clustering

\[ d = \{ f_{w_1}, f_{w_2}, \ldots, f_{w_N} \} \rightarrow z \]

**Soft clustering:** e.g., Fuzzy C-mean clustering

\[ P(Z \mid W = f_w) \]

Another representation of clustering

35
Traditional Content Clustering

**Clustering:**
Partition the feature space into *segments* based on training documents. Each segment represents a topic / category. (Topic Detection)

**Hard clustering:** e.g., K-mean clustering

\[ d = \{ f_{w_1}, f_{w_2}, \ldots, f_{w_N} \} \rightarrow z \]

**Soft clustering:** e.g., Fuzzy C-mean clustering

\[ P(Z \mid W = f_w) \]

Another representation of clustering (w/o showing the deterministic part)

Words: \( w_1, w_2, w_3, w_4, w_5, w_6 \)

Topics: \( z_1, z_2, z_3, z_4, z_5 \)

Observations: \( z_6 \)
Content Clustering based on Bayesian Network

Bayesian Network:
- Causality Network – models the causal relationship of attributes / nodes
- Allows hidden / latent nodes

Hard clustering:

\[
h(D = d) = \arg \max_z P(W = \mathbf{f}_w \mid Z) \quad \text{<= MLE}
\]

\[
P(W \mid Z) = \frac{P(Z \mid W)P(W)}{P(Z)} \quad \text{<= Bayes Theorem}
\]
Content Clustering based on Bayesian Network – Hard Clustering

Major Solution 1 -- Dirichlet Process:
- Models $P(W | Z)$ as mixtures of Dirichlet probabilities
- Before training, the prior of $P(W|Z)$ can be a easy Dirichlet (uniform distribution). After training, $P(W|Z)$ will still be Dirichlet. (The reason of using Dirichlet)

Major Solution 2 -- Gibbs Sampling:
- A Markov chain Monte Carlo (MCMC) method for integration of large samples → calculate $P(Z)$

$$P(W | Z) = \frac{P(Z | W)P(W)}{P(Z)} = \frac{P(Z | W)P(W)}{\int P(Z | W)dW}$$

$P(Z)$ shown as

$N$: the number of words
(The number of topics ($M$) are pre-determined)
Content Clustering based on Bayesian Network – Soft Clustering

- **Topics**: $z_1, z_2, z_3, z_4, z_5$
- **Words**: $w_1, w_2, w_3, w_4, w_5, w_6$
- **Documents**: $d_1, d_2, d_3$

### LDA (Blei 2003)

- **Document-Topic distributions**: $\alpha$
- **Topic-Word distributions**: $\beta$

**N**: the number of words

**A**: the number of docs
Importance of Dirichlet Distribution

• In 1982, Sandy Zabell proved that, if we make certain assumptions about an individual’s beliefs, then that individual must use the Dirichlet density function to quantify any prior beliefs about a relative frequency.
Beyond Gaussian: Examples of Dirichlet Distribution

\[ \text{Beta}(5, 1.1) \]

\[ \text{Beta}(3, 3) \]

\[ \text{Beta}(11, 5) \]

\[ \text{Beta}(0.2, 0.2) \]
Use Dirichlet Distribution to model prior and posterior beliefs

\[ \rho(f) = \text{beta}(f; a, b) \]

**beta(1,1): No prior knowledge**

**beta(3,3): prior knowledge -- this coin may be fair**
Use Dirichlet Distribution to model prior and posterior beliefs

\[ \rho(f) = \text{beta}(f; a, b) \]

\[ \rho(f \mid d) = \text{beta}(f; a + s, b + t) \]

- beta(3,3): prior knowledge -- this coin may be fair
- beta(11,5): posterior belief -- the coin may not be fair after tossing 8 heads and 2 tails
Use Dirichlet Distribution to model prior and posterior beliefs

\[ \rho(f) = \text{beta}(f; a, b) \]

\[ \rho(f \mid d) = \text{beta}(f; a + s, b + t) \]

\[ \text{beta}(1/360, 19/360): \text{prior knowledge that an 5\% chance-event may be true} \]

\[ \text{beta}(1/360+3, 19/360): \text{posterior belief that an 5\% change-event may be true} \]
Comparison of Dirichlet Distribution with Gaussian Mixture Models (1)

\[
\rho(f_1, f_2, \ldots, f_{r-1}; a_1, a_2, \ldots, a_r) = \frac{\Gamma(N)}{\prod_{k=1}^{r} \Gamma(a_k)} f_1^{a_1-1} f_2^{a_2-1} \cdots f_r^{a_r-1}
\]

\[0 \leq f_k \leq 1 \quad \sum_{k=1}^{r} f_k = 1\]

- **Multivariate Gaussian:**

\[
\rho(f_1, f_2, \ldots, f_{r-1}; \mu_1, \sigma_1, \mu_2, \sigma_2, \ldots, \mu_{r-1}, \sigma_{r-1}) = \frac{1}{(2\pi)^{r-1} \prod_{k=1}^{r-1} \sigma_k} e^{-\frac{(f_1-\mu_1)^2}{\sigma_1^2}} e^{-\frac{(f_2-\mu_2)^2}{\sigma_2^2}} \cdots e^{-\frac{(f_{r-1}-\mu_{r-1})^2}{\sigma_{r-1}^2}}
\]
**Bayesian Network:**
- Models the *practical* causal relationships.

**Content Clustering:**
- Because documents and words are dependent,
  → only close documents in the feature space can be clustered together as one topic.

⇒ Incorporating human factors can possibly *link* multiple clusters together.
Gibbs Sampling

• Suppose that it is hard to sample $p(x)$ but that it is possible to “walk around” in $X$ using local state transitions
• Insight: we can use a “random walk” to help us draw random samples from $p(x)$
• At each transition change the state of just on $X_i$
• We can describe the transition probability as a stochastic procedure:
  
  Input: a state $x_1, \ldots, x_n$
  Choose $i$ at random (using uniform probability)
  Sample $x'_i$ from
  $P(X_i | x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n, e)$
  let $x'_{j} = x_{j}$ for all $j \neq i$
  return $x'_1, \ldots, x'_n$
import org.apache.spark.ml.clustering.LDA

// Loads data.
val dataset = spark.read.format("libsvm")
  .load("data/mllib/sample_lda_libsvm_data.txt")

// Trains a LDA model.
val lda = new LDA().setK(10).setMaxIter(10)
val model = lda.fit(dataset)

val ll = model.logLikelihood(dataset)
val lp = model.logPerplexity(dataset)
println(s"The lower bound on the log likelihood of the entire corpus: $ll")
println(s"The upper bound on perplexity: $lp")

// Describe topics.
val topics = model.describeTopics(3)
println("The topics described by their top-weighted terms:")
topics.show(false)

// Shows the result.
val transformed = model.transform(dataset)
transformed.show(false)
Spark Streaming
Spark Streaming
Spark Streaming

https://www.edureka.co/blog/spark-streaming/
Spark Streaming

- **Basic Concepts**
  - Linking
  - Initializing StreamingContext
  - Discretized Streams (DStreams)
  - Input DStreams and Receivers
  - Transformations on DStreams
  - Output Operations on DStreams
  - DataFrame and SQL Operations
  - MLlib Operations
  - Caching / Persistence
  - Checkpointing
  - Accumulators, Broadcast Variables, and Checkpoints
  - Deploying Applications
  - Monitoring Applications
- **Performance Tuning**
  - Reducing the Batch Processing Times
  - Setting the Right Batch Interval
  - Memory Tuning
- **Fault-tolerance Semantics**
Spark Streaming Example

First, we import `StreamingContext`, which is the main entry point for all streaming functionality. We create a local `StreamingContext` with two execution threads, and batch interval of 1 second.

```python
from pyspark import SparkContext
from pyspark.streaming import StreamingContext

# Create a local StreamingContext with two working thread and batch interval of 1 second
sc = SparkContext("local[2]", "NetworkWordCount")
ssc = StreamingContext(sc, 1)
```

Using this context, we can create a DStream that represents streaming data from a TCP source, specified as hostname (e.g. `localhost`) and port (e.g. `9999`).

```python
# Create a DStream that will connect to hostname:port, like localhost:9999
lines = ssc.socketTextStream("localhost", 9999)
```

This lines DStream represents the stream of data that will be received from the data server. Each record in this DStream is a line of text. Next, we want to split the lines by space into words.

```python
# Split each line into words
words = lines.flatMap(lambda line: line.split(" "))
```

```python
# Count each word in each batch
pairs = words.map(lambda word: (word, 1))
wordCounts = pairs.reduceByKey(lambda x, y: x + y)
```

```python
# Print the first ten elements of each RDD generated in this DStream to the console
wordCounts.pprint()
```
Spark Streaming Example

$ ./bin/spark-submit examples/src/main/python/streaming/network_wordcount.py localhost 9999

Then, any lines typed in the terminal running the netcat server will be counted and printed on screen every second. It will look something like the following.

```
# TERMINAL 1:
# Running Netcat
$ nc -lk 9999
hello world
...
```

```
# TERMINAL 2: RUNNING network_wordcount.py

$ ./bin/spark-submit examples/src/main/python/streaming/network_wordcount.py localhost 9999
...
----------------------------------------
----------------------------------------
(hello,1)
(world,1)
...
```
Discretized Streams

DStream → RDD @ time 1 → RDD @ time 2 → RDD @ time 3 → RDD @ time 4

data from time 0 to 1 → data from time 1 to 2 → data from time 2 to 3 → data from time 3 to 4
Discretized Streams

![Diagram of Discretized Streams]

- lines DStream
  - lines from time 0 to 1
    - flatMap operation
  - lines from time 1 to 2
  - lines from time 2 to 3
  - lines from time 3 to 4

- words DStream
  - words from time 0 to 1
  - words from time 1 to 2
  - words from time 2 to 3
  - words from time 3 to 4
Discretized Streams

https://www.edureka.co/blog/spark-streaming/
DStream Transforms

https://www.edureka.co/blog/spark-streaming/
Output DStreams

Transformed DStream → Output Operations → Output DStream → External Systems

- Database
- File System

https://www.edureka.co/blog/spark-streaming/
DStreams Caching

https://www.edureka.co/blog/spark-streaming/
DStreams Example — Twitter Sentiment Analysis

```scala
// Import the necessary packages into the Spark Program
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.SparkContext._

import java.io.File

object twitterSentiment {
  def main(args: Array[String]) {
    if (args.length < 4) {
      System.err.println("Usage: TwitterPopularTags <consumer key> <consumer secret> " + ":<access token> <access token secret>
      System.exit(1)
    }
    StreamingExamples.setStreamingLogLevels()
    // Passing our Twitter keys and tokens as arguments for authorization
    val Array(consumerKey, consumerSecret, accessToken, accessTokenSecret) = args.take(4)
    val filters = args.takeRight(args.length - 4)
    // Set the system properties so that Twitter4j library used by twitter stream
    // Use them to generate OAuth credentials
    System.setProperty("twitter4j.oauth.consumerKey", consumerKey)
    System.setProperty("twitter4j.oauth.consumerSecret", consumerSecret)
    val sparkConf = new SparkConf().setAppName("twitterSentiment").setMaster("local[2]")
    val ssc = new StreamingContext
    val stream = TwitterUtils.createStream(ssc, None, filters)

    https://www.edureka.co/blog/spark-streaming/
  }
}
```
DStreams Example — Twitter Sentiment Analysis

```scala
//Input DStream transformation using flatMap
val tags = stream.flatMap { status => Get Text From The Hashtags }

//RDD transformation using sortBy and then map function
tags.countByValue()
.foreachRDD { rdd =>
  val now = Get current time of each Tweet
  rdd
    .sortBy(_._2)
    .map(x => (x, now))
  //Saving our output at ~/twitter/ directory
  .saveAsTextFile(s"~/twitter/$now")
}

//DStream transformation using filter and map functions
val tweets = stream.filter { t =>
  val tags = t. Split On Spaces .filter(_._startsWith("#")) . Convert To Lower Case
  tags.exists { x => true }
}

val data = tweets.map { status =>
  val sentiment = SentimentAnalysisUtils.detectSentiment(status.getText)
  val tagss = status.getHashtagEntities.map(_.getText.toLowerCase)
  (status.getText, sentiment.toString, tagss.toString())
}

data.print()
//Saving our output at ~/ with filenames starting like twitters
data.saveAsTextFiles("~/twitters","20000")

ssc.start()
ssc.awaitTermination()
```

https://www.edureka.co/blog/spark-streaming/
DStreams Example — Twitter Sentiment Analysis

Results:

The following are the results that are displayed in the Eclipse IDE while running the Twitter Sentiment Streaming program.

All the tweets are categorized into Positive, Neutral and Negative according to the sentiment of the contents of the tweets.

https://www.edureka.co/blog/spark-streaming/
Submission Requirements:
1. Please only submit one PDF file, containing data description, screenshots for codes and results of all questions.
2. For the dataset you have chosen, it couldn’t be too small (less than 10000 entities for example)

1. Clustering (40 points)

1-1. Choose an online news (e.g., New York Times article in 2018, Reuters News in 2018, or Yahoo News 2018) or Wikipedia, choose a dataset with category property ground-truth (which can be single layer or multi-layer-hierarchical), download the data, describe the data you use, as well as pre-processing steps if there is any (20 points)

1-2. Try three clustering algorithms provided by Spark using your data, compare the performance and discuss on the outcome of different algorithms(30 points)

2. Streaming: Twitter Sentiment Analysis (30 points)

2-1. Use Spark Streaming and Twitter API. Choose your 10 hashtags you want to monitor (10 points)

2-2. Implement your Sentiment Analysis system. Run for 3 hours. Show the outcome and discuss the performance (20 points)
3. Graph Analysis (30 points)

3-1. Calculate Similarity of the documents you downloaded at 1-1. (10 points)

3-2. Set a threshold value on the similarity to convert what you get in 3-1 into a graph. Use Spark GraphX to calculate process your graph:

- PageRank
- Connected Components
- Triangle Counting

Discuss your result and any interesting finding from the result. (20 points)
Questions?