TransmogrifAI

Salesforce

Sep 23, 2018

Contents

TransmogrifAI (pronounced trăns-mŏgr-fī) is an **AutoML** library written in Scala that runs on top of Spark. It was developed with a focus on enhancing machine learning **developer productivity** through **machine learning automation**, and an API that enforces **compile-time type-safety**, **modularity** and **reuse**.

Use TransmogrifAI if you need a machine learning library to:

- Rapidly train good quality machine learnt models with **minimal hand tuning**
- Build modular, reusable, strongly typed machine learning workflows

---

CHAPTER 1

Guiding Principles

Automation: TransmogrifAI has numerous Estimators (algorithms) that make use of TransmogrifAI feature types to automate feature engineering, feature selection, and model selection. Using these together with TransmogrifAI code-gen tools, the time taken to develop a very good model can be reduced from **several weeks** to **a couple of hours**!

Modularity and reuse: TransmogrifAI enforces a strict separation between ML workflow definitions and data manipulation, ensuring that code written using TransmogrifAI is inherently modular and reusable.

Compile-time type safety: Machine learning workflows built using TransmogrifAI are strongly typed. This means developers get to enjoy the many benefits of compile-time type safety, including code completion during development and fewer runtime errors. Workflows no longer fail several hours into model training because you tried to divide two strings!

Transparency: The type-safe nature of TransmogrifAI ensures increased transparency around inputs and outputs at every stage of your machine learning workflow. This in turn greatly reduces the amount of tribal knowledge that inevitably tends to accumulate around any sufficiently complex machine learning workflow.

CHAPTER 2

Motivation

Building real life machine learning applications needs a fair amount of tribal knowledge and intuition. Coupled with the explosion of ML use cases in the world that need to be addressed, there is a need for tools that enable rapid prototyping and development of machine learning pipelines. We believe that automation is the key to making machine learning development truly scalable and accessible.

For more information, read our blogpost!

CHAPTER 3

Documentation

3.1 Installation

- Download and install Java 1.8, then set an environment variable: `export JAVA_HOME=$(/usr/libexec/java_home -v 1.8)`
- Get Spark 2.2.x: Download, unzip it and then set an environment variable: `export SPARK_HOME=<SPARK_FOLDER>`
- Clone the TransmogrifAI repo: `git clone https://github.com/salesforce/TransmogrifAI.git`
- Build the project: `cd TransmogrifAI && ./gradlew compileTestScala installDist`
- Start hacking

3.2 Examples

3.2.1 Titanic Binary Classification

Here we describe a very simple TransmogrifAI workflow for predicting survivors in the often-cited Titanic dataset. The code for building and applying the Titanic model can be found here: Titanic Code, and the data can be found here: Titanic Data.

You can run this code as follows:

```
cd helloworld
./gradlew compileTestScala installDist
./gradlew -q sparkSubmit -Dmain=com.salesforce.hw.OpTitanicSimple -Dargs="\
`pwd`/src/main/resources/TitanicDataset/TitanicPassengersTrainData.csv"
```

Let's break down what's happening. The code starts off by describing the schema of the data via a case class:

```
// Passenger data schema
case class Passenger(
  id: Int,
  survived: Int,
  pClass: Option[Int],
  name: Option[String],
  sex: Option[String],
  age: Option[Double],
  sibSp: Option[Int],
  parCh: Option[Int],
  ticket: Option[String],
  fare: Option[Double],
  cabin: Option[String],
  embarked: Option[String]
)
```

In the main function, we create a spark session as per usual:

```
// Set up a SparkSession as normal
import org.apache.spark.SparkConf
import org.apache.spark.sql.SparkSession

val conf = new SparkConf().setAppName("TitanicPrediction")
implicit val spark = SparkSession.builder.config(conf).getOrCreate()
```

We then define the set of raw features that we would like to extract from the data. The raw features are defined using FeatureBuilders, and are strongly typed. TransmogrifAI supports the following basic feature types: Text, Numeric, Vector, List , Set, Map. In addition it supports many specific feature types which extend these base types: Email extends Text; Integral, Real and Binary extend Numeric; Currency and Percentage extend Real. For a complete view of the types supported see the Type Hierarchy and Automatic Feature Engineering section in the Documentation.

Basic FeatureBuilders will be created for you if you use the TransmogrifAI CLI to bootstrap your project as described here. However, it is often useful to edit this code to customize feature generation and take full advantage of the Feature types available (selecting the appropriate type will improve automatic feature engineering steps).

When defining raw features, specify the extract logic to be applied to the raw data, and also annotate the features as either predictor or response variables via the FeatureBuilders:

```
// import necessary packages
import com.salesforce.op.features.FeatureBuilder
import com.salesforce.op.features.types._
import com.salesforce.op._

// Define features using the TransmogrifAI types based on the data
val survived = FeatureBuilder.RealNN[Passenger].extract(_.survived.toRealNN).
↪asResponse

val pClass = FeatureBuilder.PickList[Passenger].extract(_.pClass.map(_.toString).
↪toPickList).asPredictor

val name = FeatureBuilder.Text[Passenger].extract(_.name.toText).asPredictor

val sex = FeatureBuilder.PickList[Passenger].extract(_.sex.map(_.toString).
↪toPickList).asPredictor

val age = FeatureBuilder.RealNN[Passenger].extract(_.age.toRealNN).asPredictor
```

(continues on next page)

(continued from previous page)

```
val sibSp = FeatureBuilder.Integral[Passenger].extract(_.sibSp.toIntegral).asPredictor

val parCh = FeatureBuilder.Integral[Passenger].extract(_.parCh.toIntegral).asPredictor

val ticket = FeatureBuilder.PickList[Passenger].extract(_.ticket.map(_.toString).
↪toPickList).asPredictor

val fare = FeatureBuilder.Real[Passenger].extract(_.fare.toReal).asPredictor

val cabin = FeatureBuilder.PickList[Passenger].extract(_.cabin.map(_.toString).
↪toPickList).asPredictor

val embarked = FeatureBuilder.PickList[Passenger].extract(_.embarked.map(_.toString).
↪toPickList).asPredictor
```

Now that the raw features have been defined, we go ahead and define how we would like to manipulate them via Stages (Transformers and Estimators). A TransmogrifAI Stage takes one or more Features, and returns a new Feature. TransmogrifAI provides numerous handy short cuts for specifying common feature manipulations. For basic arithmetic operations, you can just use "+", "-", "*" and "/". In addition, shortcuts like "normalize", "pivot" and "map" are also available.

```
val familySize = sibSp + parCh + 1
val estimatedCostOfTickets = familySize * fare

// normalize the numeric feature age to create a new transformed feature
val normedAge = age.zNormalize()

// pivot the categorical feature, sex, into a 0-1 vector of (male, female)
val pivotedSex = sex.pivot()

// divide age into adult and child
val ageGroup = age.map[PickList](_.value.map(v => if (v > 18) "adult" else "child").
↪toPickList)
```

The above notation is short-hand for the following, more formulaic way of invoking TransmogrifAI Stages:

```
val normedAge: FeatureLike[Numeric] = new NormalizeEstimator().setInput(age).getOutput
val pivotedSex: FeatureLike[Vector] = new PivotEstimator().setInput(sex).getOutput
```

See "Creating Shortcuts for Transformers and Estimators" for more documentation on how shortcuts for stages can be created. We now define a Feature of type Vector, that is a vector representation of all the features we would like to use as predictors in our workflow.

```
val passengerFeatures: FeatureLike[Vector] = Seq(
   pClass, name, sex, age, sibSp, parCh, ticket,
   cabin, embarked, familySize, estimatedCostOfTickets, normedAge
   pivotedSex, ageGroup
).transmogrify()
```

The `.transmogrify()` shortcut is a special AutoML Estimator that applies a default set of transformations to all the specified inputs and combines them into a single vector. This is in essence the automatic feature engineering Stage of TransmogrifAI. This stage can be discarded in favor of hand-tuned feature engineering and manual vector creation followed by combination using the VectorsCombiner Transformer (short-hand `Seq(....).combine()`) if the user desires to have complete control over feature engineering.

The next stage applies another powerful AutoML Estimator — the SanityChecker. The SanityChecker applies a variety of statistical tests to the data based on Feature types and discards predictors that are indicative of label leakage or that

show little to no predictive power. This is in essence the automatic feature selection Stage of TransmogrifAI:

```
// Optionally check the features with a sanity checker
val sanityCheck = false
val finalFeatures = if (sanityCheck) survived.sanityCheck(passengerFeatures) else␣
↪passengerFeatures
```

Finally, the OpLogisticRegression Estimator is applied to derive a new triplet of Features which are essentially probabilities and predictions returned by the logistic regression algorithm:

```
// Define the model we want to use (here a simple logistic regression) and get the␣
↪resulting output
import com.salesforce.op.stages.impl.classification.OpLogisticRegression

val prediction = new OpLogisticRegression().setInput(survived, finalFeatures).
↪getOutput
```

We could alternatively have used the ModelSelector — another powerful AutoML Estimator that automatically tries out a variety of different classification algorithms and then selects the best one.

Notice that everything we've done so far has been purely at the level of definitions. We have defined how we would like to extract our raw features from data of type 'Passenger', and we have defined how we would like to manipulate them. In order to actually manifest the data described by these features, we need to add them to a workflow and attach a data source to the workflow:

```
import com.salesforce.op.readers.DataReaders

val trainDataReader = DataReaders.Simple.csvCase[Passenger](
      path = Some(csvFilePath), // location of data file
      key = _.id.toString  // identifier for entity being modeled
)

val workflow =
   new OpWorkflow()
      .setResultFeatures(survived, prediction)
      .setReader(trainDataReader)
```

When we now call 'train' on this workflow, it automatically computes and executes the entire DAG of Stages needed to compute the features `survived`, `prediction`, `rawPrediction`, and `prob`, fitting all the estimators on the training data in the process. Calling `score` on the fitted workflow then transforms the underlying training data to produce a DataFrame with the all the features manifested. The `score` method can optionally be passed an evaluator that produces metrics.

```
// Fit the workflow to the data
val fittedWorkflow = workflow.train()

val evaluator = Evaluators.BinaryClassification()
   .setLabelCol(survived)
   .setPredictionCol(prediction)

// Apply the fitted workflow to the train data and manifest
// the resulting dataframe together with metrics
val (transformedTrainData, metrics) = fittedWorkflow.scoreAndEvaluate(evaluator =␣
↪evaluator)
```

The fitted workflow can now be saved, and loaded again to be applied to any new data set of type Passengers by changing the reader.

```
fittedWorkflow.save(saveWorkflowPath)

val savedWorkflow = workflow.loadModel(saveWorkflowPath).setReader(testDataReader)
```

3.2.2 Iris MultiClass Classification

The following code illustrates how TransmogrifAI can be used to do classify multiple classes over the Iris dataset. The code for this example can be found here, and the data over here.

Define features

```
val id = FeatureBuilder.Integral[Iris].extract(_.getID.toIntegral).asPredictor
val sepalLength = FeatureBuilder.Real[Iris].extract(_.getSepalLength.toReal).
↪asPredictor
val sepalWidth = FeatureBuilder.Real[Iris].extract(_.getSepalWidth.toReal).asPredictor
val petalLength = FeatureBuilder.Real[Iris].extract(_.getPetalLength.toReal).
↪asPredictor
val petalWidth = FeatureBuilder.Real[Iris].extract(_.getPetalWidth.toReal).asPredictor
val irisClass = FeatureBuilder.Text[Iris].extract(_.getClass$.toText).asResponse
```

Feature Engineering

```
val labels = irisClass.indexed()
val features = Seq(sepalLength, sepalWidth, petalLength, petalWidth).transmogrify()
```

Modeling & Evaluation

```
val pred = MultiClassificationModelSelector
  .withCrossValidation(splitter = Some(DataCutter(reserveTestFraction = 0.2, seed =␣
↪randomSeed)), seed = randomSeed)
  .setInput(labels, features).getOutput()

private val evaluator = Evaluators.MultiClassification.f1()
  .setLabelCol(labels)
  .setPredictionCol(pred)

private val wf = new OpWorkflow().setResultFeatures(pred, labels)

def runner(opParams: OpParams): OpWorkflowRunner =
  new OpWorkflowRunner(
    workflow = wf,
    trainingReader = irisReader,
    scoringReader = irisReader,
    evaluationReader = Option(irisReader),
    evaluator = Option(evaluator),
    featureToComputeUpTo = Option(features)
  )
```

You can run the code using the following commands for train, score and evaluate:

```
cd helloworld
./gradlew compileTestScala installDist
```

Train

```
./gradlew -q sparkSubmit -Dmain=com.salesforce.hw.iris.OpIris -Dargs="\
--run-type=train \
--model-location=/tmp/iris-model \
--read-location Iris=`pwd`/src/main/resources/IrisDataset/iris.data"
```

Score

```
./gradlew -q sparkSubmit -Dmain=com.salesforce.hw.iris.OpIris -Dargs="\
--run-type=score \
--model-location=/tmp/iris-model \
--read-location Iris=`pwd`/src/main/resources/IrisDataset/bezdekIris.data \
--write-location=/tmp/iris-scores"
```

Evaluate

```
./gradlew -q sparkSubmit -Dmain=com.salesforce.hw.iris.OpIris -Dargs="\
--run-type=evaluate \
--model-location=/tmp/iris-model \
--metrics-location=/tmp/iris-metrics \
--read-location Iris=`pwd`/src/main/resources/IrisDataset/bezdekIris.data \
--write-location=/tmp/iris-eval"
```

3.2.3 Boston Regression

The following code illustrates how TransmogrifAI can be used to do linear regression. We use Boston dataset to predict housing prices. The code for this example can be found here, and the data over here.

Define features

```
val rowId = FeatureBuilder.Integral[BostonHouse].extract(_.rowId.toIntegral).
↪asPredictor
val crim = FeatureBuilder.RealNN[BostonHouse].extract(_.crim.toRealNN).asPredictor
val zn = FeatureBuilder.RealNN[BostonHouse].extract(_.zn.toRealNN).asPredictor
val indus = FeatureBuilder.RealNN[BostonHouse].extract(_.indus.toRealNN).asPredictor
val chas = FeatureBuilder.PickList[BostonHouse].extract(x => Option(x.chas).
↪toPickList).asPredictor
val nox = FeatureBuilder.RealNN[BostonHouse].extract(_.nox.toRealNN).asPredictor
val rm = FeatureBuilder.RealNN[BostonHouse].extract(_.rm.toRealNN).asPredictor
val age = FeatureBuilder.RealNN[BostonHouse].extract(_.age.toRealNN).asPredictor
val dis = FeatureBuilder.RealNN[BostonHouse].extract(_.dis.toRealNN).asPredictor
val rad = FeatureBuilder.Integral[BostonHouse].extract(_.rad.toIntegral).asPredictor
val tax = FeatureBuilder.RealNN[BostonHouse].extract(_.tax.toRealNN).asPredictor
val ptratio = FeatureBuilder.RealNN[BostonHouse].extract(_.ptratio.toRealNN).
↪asPredictor
val b = FeatureBuilder.RealNN[BostonHouse].extract(_.b.toRealNN).asPredictor
val lstat = FeatureBuilder.RealNN[BostonHouse].extract(_.lstat.toRealNN).asPredictor
val medv = FeatureBuilder.RealNN[BostonHouse].extract(_.medv.toRealNN).asResponse
```

Feature Engineering

```
val houseFeatures = Seq(crim, zn, indus, chas, nox, rm, age, dis, rad, tax, ptratio,␣
↪b, lstat).transmogrify()
```

Modeling & Evaluation

```
val prediction = RegressionModelSelector
  .withCrossValidation(dataSplitter = Option(DataSplitter(seed = randomSeed)), seed =
↪randomSeed)
  .setRandomForestSeed(randomSeed)
  .setGradientBoostedTreeSeed(randomSeed)
  .setInput(medv, houseFeatures)
  .getOutput()

val workflow = new OpWorkflow().setResultFeatures(prediction)

val evaluator = Evaluators.Regression().setLabelCol(medv).setPredictionCol(prediction)

def runner(opParams: OpParams): OpWorkflowRunner =
  new OpWorkflowRunner(
    workflow = workflow,
    trainingReader = trainingReader,
    scoringReader = scoringReader,
    evaluationReader = Option(trainingReader),
    evaluator = Option(evaluator),
    scoringEvaluator = None,
    featureToComputeUpTo = Option(houseFeatures)
  )
```

You can run the code using the following commands for train, score and evaluate:

```
cd helloworld
./gradlew compileTestScala installDist
```

Train

```
./gradlew -q sparkSubmit -Dmain=com.salesforce.hw.boston.OpBoston -Dargs="\
--run-type=train \
--model-location=/tmp/boston-model \
--read-location BostonHouse=`pwd`/src/main/resources/BostonDataset/housing.data"
```

Score

```
./gradlew -q sparkSubmit -Dmain=com.salesforce.hw.boston.OpBoston -Dargs="\
--run-type=score \
--model-location=/tmp/boston-model \
--read-location BostonHouse=`pwd`/src/main/resources/BostonDataset/housing.data \
--write-location=/tmp/boston-scores"
```

Evaluate

```
./gradlew -q sparkSubmit -Dmain=com.salesforce.hw.boston.OpBoston -Dargs="\
--run-type=evaluate \
--read-location BostonHouse=`pwd`/src/main/resources/BostonDataset/housing.data \
--write-location=/tmp/boston-eval \
--model-location=/tmp/boston-model \
--metrics-location=/tmp/boston-metrics"
```

3.2.4 Time Series Aggregates and Joins

In this example, we will walk you through some of the powerful tools TransmogrifAI has for data preparation, in particular for time series aggregates and joins. The code for this example can be found here, and the data over here.

In this example, we would like to build a training data set from two different tables – a table of Email Sends, and a table of Email Clicks. The following case classes describe the schemas of the two tables:

```
case class Click(clickId: Int, userId: Int, emailId: Int, timeStamp: String)
case class Send(sendId: Int, userId: Int, emailId: Int, timeStamp: String)
```

The goal is to build a model that will predict the number of times a user will click on emails on day `x+1`, given his click behavior in the lead-up to day `x`. The ideal training dataset would be constructed by taking a certain point in time as a reference point. And then for every user in the tables, computing a response that is the number of times the user clicked on an email within a day of that reference point. The features for every user would be computed by aggregating his click behavior up until that reference point.

Unlike the previous examples, these tables represent events – a single user may have been sent multiple emails, or clicked on multiple emails, and the events need to be aggregated in order to produce meaningful predictors and response variables for a training data set.

TransmogrifAI provides an easy way for us to define these aggregate features. Using a combination of FeatureBuilders and Aggregate Readers. Let's start with the readers. We define two readers for the two different tables as follows:

```
val clicksReader = DataReaders.Aggregate.csvCase[Click](
   path = Some("src/main/resources/EmailDataset/Clicks.csv"),
   key = _.userId.toString,
   aggregateParams = AggregateParams(
      timeStampFn = Some[Click => Long](c => formatter.parseDateTime(c.timeStamp).
↪getMillis),
      cutOffTime = CutOffTime.DDMMYYYY("04092017")
   )
)

val sendsReader = DataReaders.Aggregate.csvCase[Send](
   path = Some("src/main/resources/EmailDataset/Sends.csv"),
   key = _.userId.toString,
   aggregateParams = AggregateParams(
      timeStampFn = Some[Send => Long](s => formatter.parseDateTime(s.timeStamp).
↪getMillis),
      cutOffTime = CutOffTime.DDMMYYYY("04092017")
   )
)
```

There are a few different parameters of interest in these readers:

- The first is a `key` parameter, that specifies the key in the table that should be used to aggregate either the predictors or response variables.
- The second is a `timeStampFn` parameter that allows the user to specify a function for extracting timestamps from records in the table. This is the timestamp that will be used to compare against the reference time.
- And the third is a `cutOffTime`, which is the reference time to be used. All predictors will be aggregated from records up until the `cutOffTime`, and all response variables will be aggregated from records following the `cutOffTime`.

Now let's look at how the predictors and response variables are defined. In this example, we define two aggregate predictors using TransmogrifAI's FeatureBuilders:

```
val numClicksYday = FeatureBuilder.RealNN[Click]
    .extract(click => 1.toRealNN)
    .aggregate(SumRealNN)
    .window(Duration.standardDays(1))
    .asPredictor
```

(continues on next page)

(continued from previous page)

```
val numSendsLastWeek = FeatureBuilder.RealNN[Send]
    .extract(send => 1.toRealNN)
    .aggregate(SumRealNN)
    .window(Duration.standardDays(7))
    .asPredictor
```

Here `numClicksYday` is a non-nullable real predictor, extracted from the Clicks table, by mapping each click to a `1`, then aggregating for each key of the Click table by summing up the `1's` that occur in a 1 day window before the `cutOffTime` specified in the `clicksReader`.

Similarly, `numSendsLastWeek` is obtained by aggregating for each key of the Send table, all the sends that occur in a 7 day windown prior to the `cutOffTime` specified in the `sendsReader`.

The response variable on the other hand, is obtained by aggregating all the clicks that occur in a 1 day window following the `cutOffTime` specified in the `clicksReader`:

```
val numClicksTomorrow = FeatureBuilder.RealNN[Click]
    .extract(click => 1.toRealNN)
    .aggregate(SumRealNN)
    .window(Duration.standardDays(1))
    .asResponse
```

Now we can also create a predictor from the combination of the clicks and sends predictors as follows:

```
// .alias ensures that the resulting dataframe column name is 'ctr'
// and not the default transformed feature name
val ctr = (numClicksYday / (numSendsLastWeek + 1)).alias
```

In order to materialize all of these predictors and response variables, we can add them to a workflow with the appropriate readers:

```
// fit the workflow to the data
val workflowModel = new OpWorkflow()
    .setReader(sendsReader.leftOuterJoin(clicksReader))
    .setResultFeatures(numClicksYday, numClicksTomorrow, numSendsLastWeek, ctr)
    .train()

// materialize the features
val dataFrame = workflowModel.score()
```

Note that the reader for the workflow is a joined reader, obtained by joining the `sendsReader` with the `clicksReader`. The joined reader deals with nulls in the two tables appropriately:

```
dataFrame.show()

+---+---+-----------------+-------------+----------------+
|ctr|key|numClicksTomorrow|numClicksYday|numSendsLastWeek|
+---+---+-----------------+-------------+----------------+
|0.0|789|             null|         null|             1.0|
|0.0|456|              1.0|          0.0|             0.0|
|1.0|123|              1.0|          2.0|             1.0|
+---+---+-----------------+-------------+----------------+
```

3.2.5 Conditional Aggregation

In this example, we demonstrate use of TransmogrifAI's conditional readers to, once again, simplify complex data preparation. Code for this example can be found here, and the data can be found here.

In the previous example, we showed how TransmogrifAI FeatureBuilders and Aggregate Readers could be used to aggregate predictors and response variables with respect to a reference point in time. However, sometimes, aggregations need to be computed with respect to the time of occurrence of a particular event, and this time may vary from key to key. In particular, let's consider a situation where we are analyzing website visit data, and would like to build a model that predicts the number of purchases a user makes on the website within a day of visiting a particular landing page. In this scenario, we need to construct a training dataset that for each user, identifies the time when he visited the landing page, and then creates a response which is the number of times the user made a purchase within a day of that time. The predictors for the user would be aggregated from the web visit behavior of the user up unto that point in time.

Let's start once again by looking at the reader. The web visit data is described by the following case class:

```
case class WebVisit(
    userId: String,
    url: String,
    productId: Option[Int],
    price: Option[Double],
    timestamp: String
)
```

We read this data using a Conditional Aggregate Reader:

```
val visitsReader = DataReaders.Conditional.csvCase[WebVisit](
    path = Some("src/main/resources/WebVisitsDataset/WebVisits.csv"),
    key = _.userId,
    conditionalParams = ConditionalParams(
        timeStampFn = visit => formatter.parseDateTime(visit.timestamp).getMillis,
        targetCondition = _.url == "http://www.amazon.com/SaveBig",
        dropIfTargetConditionNotMet = true
    )
)
```

Once again, there are a few different parameters of note in this reader.

- The `key` specifies the key in the table that should be used to aggregate the predictors or response variables
- The `targetCondition` specifies the function to be applied to a record to see if the target condition is met. In this case, the event of interest is whether the user visited the Amazon Save Big landing page.
- The `timeStampFn` provides the function to be applied to a record to extract its timestamp and compare to the timestamp of the target event.
- `dropIfTargetConditionNotMet` when set to `true` drops all keys where the target condition was not met.

The predictor and response variables are specified as before:

```
val numVisitsWeekPrior = FeatureBuilder.RealNN[WebVisit]
    .extract(visit => 1.toRealNN)
    .aggregate(SumRealNN)
    .window(Duration.standardDays(7))
    .asPredictor

val numPurchasesNextDay = FeatureBuilder.RealNN[WebVisit]
    .extract(visit => visit.productId.map{_ => 1D}.toRealNN)
```

(continues on next page)

(continued from previous page)

```
    .aggregate(SumRealNN)
    .window(Duration.standardDays(1))
    .asResponse
```

And finally, the predictors, response variables, and readers are all fed to a workflow and the training dataset is materialized:

```
val workflowModel = new OpWorkflow()
    .setReader(visitsReader)
    .setResultFeatures(numVisitsWeekPrior, numPurchasesNextDay)
    .train()

val dataFrame = workflowModel.score()
```

The TransmogrifAI workflow automatically identifies when the target condition was met for each key in the table, and aggregates the predictor and response variables for each appropriately:

```
dataFrame.show()

+-----------------+-------------------+-----------------+
|              key|numPurchasesNextDay|numVisitsWeekPrior|
+-----------------+-------------------+-----------------+
|xyz@salesforce.com|                1.0|              3.0|
|lmn@salesforce.com|                1.0|              0.0|
|abc@salesforce.com|                0.0|              1.0|
+-----------------+-------------------+-----------------+
```

3.2.6 Running from Spark Shell

Start up your spark shell and add the TransmogrifAI package:

```
$SPARK_HOME/bin/spark-shell --packages com.salesforce.transmogrifai:transmogrifai-
↪core_2.11:0.4.0
```

Or if you'd like to use the latest version from master:

```
cd TransmogrifAI && ./gradlew core:shadowJar
$SPARK_HOME/bin/spark-shell --jars core/build/libs/transmogrifai-x.y.z-all.jar
```

Once the `spark-shell` starts up, create your spark session:

```
// Use the existing Spark session
implicit val spark = ss
// or set up a new one SparkSession if needed
import org.apache.spark.SparkConf
import org.apache.spark.sql.SparkSession

val conf = new SparkConf().setAppName("TitanicPrediction")
implicit val spark = SparkSession.builder.config(conf).getOrCreate()
```

Import TransmogrifAI:

```
// All the TransmogrifAI functionality: feature types, feature builders, feature dsl,␣
↪readers, aggregators etc.
import com.salesforce.op._
```

(continues on next page)

(continued from previous page)

```
import com.salesforce.op.readers._
import com.salesforce.op.features._
import com.salesforce.op.features.types._
import com.salesforce.op.aggregators._

// Optional - Spark type enrichments as follows
import com.salesforce.op.utils.spark.RichDataset._
import com.salesforce.op.utils.spark.RichRDD._
import com.salesforce.op.utils.spark.RichMetadata._
import com.salesforce.op.utils.spark.RichStructType._
```

Now follow along with the rest of the code from the Titanic example found here.

3.2.7 Bootstrap Your First Project

We provide a convenient way to bootstrap you first project with TransmogrifAI using the TransmogrifAI CLI. As an illustration, let's generate a binary classification model with the Titanic passenger data.

Clone the TransmogrifAI repo:

```
git clone https://github.com/salesforce/TransmogrifAI.git
```

Checkout the latest release branch (in this example `0.4.0`):

```
cd ./TransmogrifAI
git checkout 0.4.0
```

Build the TransmogrifAI CLI by running:

```
./gradlew cli:shadowJar
alias transmogrifai="java -cp `pwd`/cli/build/libs/\* com.salesforce.op.cli.CLI"
```

Finally generate your Titanic model project (follow the instructions on screen):

```
transmogrifai gen --input `pwd`/test-data/PassengerDataAll.csv \
  --id passengerId --response survived \
  --schema `pwd`/test-data/PassengerDataAll.avsc Titanic
```

If you run this command more than once, two important command line arguments will be useful:

- `--overwrite` will allow to overwrite an existing project; if not specified, the generator will fail
- `--answers <answers_file>` will provide answers to the questions that the generator asks.

e.g.

```
transmogrifai gen --input `pwd`/test-data/PassengerDataAll.csv \
  --id passengerId --response survived \
  --schema `pwd`/test-data/PassengerDataAll.avsc \
  --answers cli/passengers.answers Titanic --overwrite
```

will do the generation without asking you anything.

Here we have specified the schema of the input data as an Avro schema. Avro is the schema format that the TransmogrifAI CLI understands. Note that when writing up your machine learning workflow by hand, you can always use case classes instead.

Your Titanic model project is ready to go.

You will notice a default set of FeatureBuilders generated from the provided Avro schema. You are encouraged to edit this code to customize feature generation and take full advantage of the Feature types available (selecting the appropriate type will improve automatic feature engineering steps).

The generated code also uses the `.transmogrify()` shortcut to apply default feature transformations to the raw features and create a single feature vector. This is in essence the automatic feature engineering Stage of TransmogrifAI. Once again, you can customize and expand on the feature manipulations applied by acting directly on individual features before applying `.vectorize()`. You can also choose to completely discard `.vectorize()` in favor of hand-tuned feature engineering and manual vector creation using the VectorsCombiner Estimator (short-hand `Vectorizers.combine()`) if you desire to have complete control over the feature engineering.

For convenience we have provided a simple `OpAppWithRunner` (and a more customizable `OpApp`) which takes in a workflow and allows you to run spark jobs from the command line rather than creating your own Spark App.

```
object Titanic extends OpAppWithRunner with TitanicWorkflow {
   def runner(opParams: OpParams): OpWorkflowRunner =
    new OpWorkflowRunner(
      workflow = titanicWorkflow,
      trainingReader = trainingReader,
      scoringReader = scoringReader,
      evaluator = evaluator,
      scoringEvaluator = None,
      featureToComputeUpTo = featureVector,
      kryoRegistrator = classOf[TitanicKryoRegistrator]
    )
}
```

This app is generated as part of the template and can be run like this:

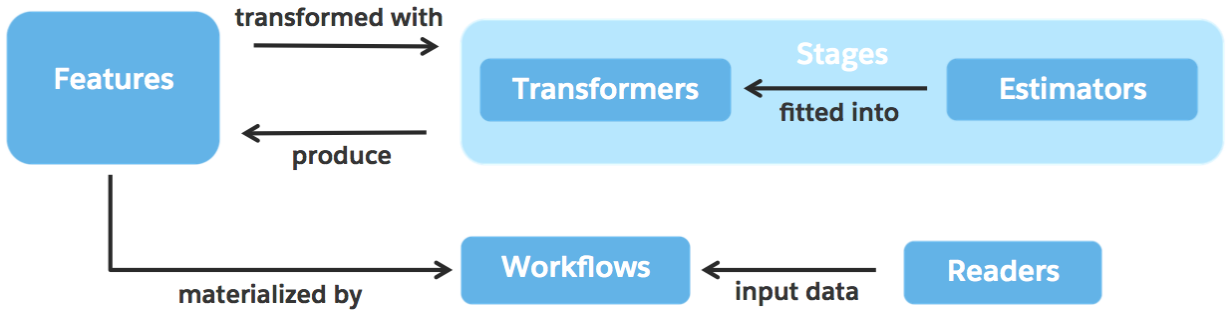
```
cd titanic
./gradlew compileTestScala installDist
./gradlew sparkSubmit -Dmain=com.salesforce.app.Titanic -Dargs="--run-type=train --
↪model-location=/tmp/titanic-model --read-location Passenger=`pwd`/../test-data/
↪PassengerDataAll.csv"
```

To generate a project for any other dataset, simply modify the parameters to point to your specific data and its schema.

Happy modeling!

3.3 Abstractions

TransmogrifAI is designed to simplify the creation of machine learning workflows. To this end we have created an abstraction for creating and running machine learning workflows. The abstraction is made up of Features, Stages, Workflows and Readers which interact as shown in the diagram below.



3.3.1 Features

The primary abstraction introduced in TransmogrifAI is that of a Feature. A Feature is essentially a type-safe pointer to a column in a DataFrame and contains all information about that column – it's name, the type of data to be found in it, as well as lineage information about how it was derived. Features are defined using FeatureBuilders:

```
val name: Feature[Text] = FeatureBuilder.Text[Passenger].extract(_.name.toText).
↪asPredictor
val age: Feature[RealNN] = FeatureBuilder.RealNN[Passenger].extract(_.age.toRealNN).
↪asPredictor
```

The above lines of code define two `Features` of type `Text` and `RealNN` called `name` and `age` that are extracted from data of type `Passenger` by applying the stated extract methods.

One can also define Features that are the result of complex time-series aggregates. Take a look at this example and this page for more advanced reading on FeatureBuilders.

Features can then be manipulated using Stages to produce new Features. In TransmogrifAI, as in SparkML, there are two types of Stages – Transformers and Estimators.

3.3.2 Stages

3.3.2.1 Transformers

Transformers specify functions for transforming one or more Features to one or more *new* Features. Here is an example of applying a tokenizing Transformer to the `name` Feature defined above:

```
val nameTokens = new TextTokenizer[Text]().setAutoDetectLanguage(true).setInput(name).
↪getOutput()
```

The output `nameTokens` is a new Feature of type `TextList`. Because Features are strongly typed, it is also possible to create shortcuts for these Transformers and create a Feature operation syntax. The above line could alternatively have been written as:

```
val nameTokens = name.tokenize()
```

TransmogrifAI provides an easy way for wrapping all Spark Transformers, and additionally provides many Transformers of its own. For more reading about creating new Transformers and shortcuts, follow the links here and here.

3.3.2.2 Estimators

Estimators specify algorithms that can be applied to one or more Features to produce Transformers that in turn produce new Features. Think of Estimators as learning algorithms, that need to be fit to the data, in order to then be able to transform it. Users of TransmogrifAI do not need to worry about the fitting of algorithms, this happens automatically behind the scenes when a TransmogrifAI workflow is trained. Below we see an example of a use of a bucketizing estimator that determines the buckets that maximize information gain when fit to the data, and then transforms the Feature `age` to a new bucketized Feature of type `OPVector`:

```
val bucketizedAge = new DecisionTreeNumericBucketizer[Double, Real]().setInput(label,␣
↪age).getOutput()
```

Similar to Transformers above, one can easily create shortcuts for Estimators, and so the line of code above could have been alternatively written as:

```
val bucketizedAge = age.autoBucketize(label = label)
```

TransmogrifAI provides an easy way for wrapping all Spark Estimators, and additionally provides many Estimators of its own. For more reading about creating new Estimators follow the link here.

3.3.3 Workflows and Readers

Once all the Features and Feature transformations have been defined, actual data can be materialized by adding the desired Features to a TransmogrifAI Workflow and feeding it a DataReader. When the Workflow is trained, it infers the entire DAG of Features, Transformers, and Estimators that are needed to materialize the result Features. It then prepares this DAG by passing the data specified by the DataReader through the DAG and fitting all the intermediate Estimators in the DAG to Transformers.

In the example below, we would like to materialize `bucketizedAge` and `nameTokens`. So we set these two Features as the result Features for a new Workflow:

```
val workflow = new OPWorkflow().setResultFeatures(bucketizedAge, nameTokens).
↪setReader(PassengerReader)
```

The PassengerReader is a DataReader that essentially specifies a `read` method that can be used for loading the Passenger data. When we train this workflow, it reads the Passenger data and fits the bucketization estimator by determining the optimal buckets for `age`:

```
val workflowModel = workflow.train()
```

The workflowModel now has a prepped DAG of Transformers. By calling the `score` method on the workflowModel, we can transform any data of type Passenger to a DataFrame with two columns for `bucketizedAge` and `nameTokens`

```
val dataFrame = workflowModel.setReader(OtherPassengerReader).score()
```

WorkflowModels can be saved and loaded. For more advanced reading on topics like stacking workflows, aggregate DataReaders for time-series data, or joins for DataReaders, follow our links to Workflows and Readers.

3.4 AutoML Capabilities

3.4.1 Vectorizers and Transmogrification

This is the Stage that automates the feature engineering step in the machine learning pipeline.

The TransmogrifAI transmogrifier (shortcut `.transmogrify()`) takes in a sequence of features, automatically applies default transformations to them based on feature types (e.g. imputation, null value tracking, one hot encoding, tokenization, split Emails and pivot out the top K domains) and combines them into a single vector.

```
val features = Seq(email, phone, age, subject, zipcode).transmogrify()
```

If you want to do the feature engineering at a single feature level, you can do so in combination with automatic type specific transformations. Each feature type has an associated `.vectorize(....)` method that will transform the feature into a feature vector given some input parameters. Each `.vectorize(....)` method behaves differently according to the type of feature being transformed.

```
val emailFeature = email.vectorize()
val features = Seq(emailFeature, phone, age, subject, zipcode).transmogrify()
```

For advanced users, you can also completely customize automatic feature engineering.

3.4.2 Feature Validation

3.4.2.1 SanityChecker

This is the Stage that automates the feature selection step in the machine learning pipeline.

The SanityChecker is an Estimator that can analyze a particular dataset for obvious issues prior to fitting a model on it. It applies a variety of statistical tests to the data based on Feature types and discards predictors that are indicative of label leakage or that show little to no predictive power. In addition to flagging and fixing data issues, the SanityChecker also outputs statistics about the data for diagnostics and insight generation further down the ML pipeline.

The SanityChecker can be instantiated as follows:

```
// Add sanity checker estimator
val checkedFeatures = new SanityChecker().setRemoveBadFeatures(true).setInput(label,␣
↪features).getOutput()
```

For advanced users, check out how to customize default parameters and peek into the SanityChecker metadata using model insights.

3.4.2.2 RawFeatureFilter

One of the fundamental assumptions of machine learning is that the data you are using to train your model reflects the data that you wish to score. In the real world, this assumption is often not true. TransmogrifAI has an optional stage after data reading that allows you to check that your features do not violate this assumption and remove any features that do. This stage is called the RawFeatureFilter, and to use it you call the method `withRawFeatureFilter(Option(trainReader), Option(scoreReader),...)` on your Workflows. This method takes the training and scoring data readers as inputs.

```
// Add raw feature filter estimator
val workflow =
   new OpWorkflow()
      .setResultFeatures(survived, rawPrediction, prob, prediction)
      .withRawFeatureFilter(Option(trainReader), Option(scoreReader), None)
```

It will load the training and scoring data and exclude individual features based on fill rate, relative fill rates between training and scoring, or differences in the distribution of data between training and scoring. This stage can eliminate many issues, such as leakage of information that is only filled out after the label and changes in data collection practices, before they affect your model.

For advanced users, check out how to set optional parameters for when to exclude features.

3.4.3 ModelSelectors

This is the Stage that automates the model selection step in the machine learning pipeline.

TransmogrifAI will select the best model and hyper-parameters for you based on the class of modeling you are doing (eg. Classification, Regression etc.). Smart model selection and comparison gives the next layer of improvements over traditional ML workflows.

```
val pred = BinaryClassificationModelSelector().setInput(label, features).getOutput()
```

The ModelSelector is an Estimator that uses data to find the best model. BinaryClassificationModelSelector is for binary classification tasks, multi classification tasks can be done using MultiClassificationModelSelector. Best Regression model are done through RegressionModelSelector. Currently the possible classification models that can be applied in the selector are `GBTCLassifier`, `LinearSVC`, `LogisticRegression`, `DecisionTrees`, `RandomForest` and `NaiveBayes`, though `GBTClassifier` and `LinearSVC` only support binary classification. The possible regression models are `GeneralizedLinearRegression`, `LinearRegression`, `DecisionTrees`, `RandomForest` and `GBTreeRegressor`. The best model is selected via a CrossValidation or TrainingSplit, by picking the best model and wrapping it. By default each of these models comes with a predefined set of hyperparameters that will be tested in determining the best model.

For advanced users, check out how to specify specific models and hyperparameters, add your own models, set validation parameters, and balance datasets here.

3.5 FAQ

3.5.1 What is TransmogrifAI?

TransmogrifAI is an AutoML library written in Scala that runs on top of Spark. It was developed with a focus on enhancing machine learning developer productivity through machine learning automation, and an API that enforces compile-time type-safety, modularity and reuse.

Use TransmogrifAI if you need a machine learning library to:

- Rapidly train good quality machine learnt models with minimal hand tuning
- Build modular, reusable, strongly typed machine learning workflows

3.5.2 Why is "op" in the package name and at the start of many class names?

OP is a reference to the internal codename that the project was developed under: Optimus Prime.

3.5.3 I am used to working in Python why should I care about type safety?

The flexibility of Salesforce Objects allows customers to modify even standard objects schemas. This means that when writing models for a multi-tenant environment the only information about what is in a column that we can really count on is the Salesforce type (i.e. Phone, Email, Mulipicklist, Percent, etc.). Working in a strictly typed environment allows us to leverage this information to perform sensible automatic feature engineering.

In addition type safety assures that you get fewer unexpected data issues in production.

3.5.4 What does automatic feature engineering based on types look like?

In order to take advantage of automatic type based feature engineering in TransmogrifAI one simply defines the features that will be used in the model and relies on TransmogrifAI to do the feature engineering. The code for this would look like:

```
val featureVector = Seq(email, name, description, salary, phone).transmogrify()
```

The transmogrify shortcut will sort the features by type and apply appropriate transformations. For example, email which will have the type `Email` will be split into prefix and domain, checked for spam-i-ness and pivoted for top domains. Similarly, description which will have the type `TextArea` will be automatically be converted to a feature vector using the hashing trick.

Of course if you want to manually perform these or other transformations you can simply specify the steps for each feature and use the VectorsCombiner Transformer to manually combine your final features. However, this gives developers the option of using default type specific feature engineering.

3.5.5 What other AutoML functionality does TransmogrifAI provide?

Look at the AutoML Capabilities section for a complete list of the powerful AutoML estimators that TransmogrifAI provides. In a nutshell, they are Transmogrifier for automatic feature engineering, SanityChecker and RawFeatureFilter for data cleaning and automatic feature selection, and ModelSelectors for different classes of problems for automatic model selection.

3.5.6 What imports do I need for TransmogrifAI to work?

```
// TransmogrifAI functionality: feature types, feature builders, feature dsl, readers,
↪ aggregators etc.
import com.salesforce.op._
import com.salesforce.op.aggregators._
import com.salesforce.op.features._
import com.salesforce.op.features.types._
import com.salesforce.op.readers._

// Spark enrichments (optional)
import com.salesforce.op.utils.spark.RichDataset._
import com.salesforce.op.utils.spark.RichRDD._
import com.salesforce.op.utils.spark.RichRow._
import com.salesforce.op.utils.spark.RichMetadata._
import com.salesforce.op.utils.spark.RichStructType._
```

3.5.7 I don't need joins or aggregations in my data preparation why can't I just use Spark to load my data and pass it into a Workflow?

You can! Simply use the `.setInputRDD(myRDD)` or `.setInputDataSet(myDataSet)` methods on Workflow to pass in your data.

3.5.8 How do I examine intermediate data when trying to debug my ML workflow?

You can generate data up to any particular point in the Workflow using the method `.computeDataUpTo(myFeature)`. Calling this method on your Workflow or WorkflowModel will compute a DataFrame which contains all of the rows for features created up to that point in your flow.

3.6 Talks

2018

- AutoML: The Assembly Line of Machine Learning, Mayukh Bhaowal, DataEngConf
- The Black Swan of Perfectly Interpretable Models, Leah McGuire and Mayukh Bhaowal, QCon.ai
- Implementing AutoML Techniques at Salesforce Scale, Matthew Tovbin, Spark+AI Summit, Slides

2017

- Embracing a Taxonomy of Types to Simplify Machine Learning, Leah McGuire, Spark Summit, Slides
- When all the world's data scientists are just not enough, Shubha Nabar, The @Scale Conference
- Low Touch Machine Learning, Leah McGuire, Spark Summit
- Fantastic ML apps and how to build them, Matthew Tovbin, Scale By The Bay, Slides

2016

- Metadata Science: When the world's data scientists are not enough, Shubha Nabar, Scala By The Bay
- Doubt Truth to be a Liar: Non Triviality of Type Safety for Machine Learning, Matthew Tovbin, Scala By The Bay / Scala Days, Slides

3.7 Contributing

This page lists recommendations and requirements for how to best contribute to TransmogrifAI. We strive to obey these as best as possible. As always, thanks for contributing – we hope these guidelines make it easier and shed some light on our approach and processes.

3.7.1 Issues, requests & ideas

Use GitHub Issues page to submit issues, enhancement requests and discuss ideas.

3.7.2 Contributing

1. **Ensure the bug/feature was not already reported** by searching on GitHub under Issues. If none exists, create a new issue so that other contributors can keep track of what you are trying to add/fix and offer suggestions (or let you know if there is already an effort in progress).
2. **Clone** the forked repo to your machine.
3. **Commit** changes to your own branch.
4. **Push** your work back up to your fork.
5. **Submit** a Pull Request against the `master` branch and refer to the issue(s) you are fixing. Try not to pollute your pull request with unintended changes. Keep it simple and small.

   NOTE: Be sure to sync your fork before making a pull request.

3.7.3 Contribution Checklist

- Clean, simple, well styled code
- Comments
  - Module-level & function-level comments.
  - Comments on complex blocks of code or algorithms (include references to sources).
- Tests
  - Increase code coverage, not versa.
  - Use ScalaTest with `FlatSpec` and `PropSpec`.

  - Use our testkit that contains a bunch of testing facilities you would need. Simply `import com.salesforce.op.test._` and borrow inspiration from existing tests.

- Dependencies
  - Minimize number of dependencies.
  - Prefer BSD, Apache 2.0, MIT, ISC and MPL licenses.

3.7.4 Code of Conduct

Follow the Apache Code of Conduct.

3.7.5 License

By contributing your code, you agree to license your contribution under the terms of the BSD 3-Clause.

3.8 Developer Guide

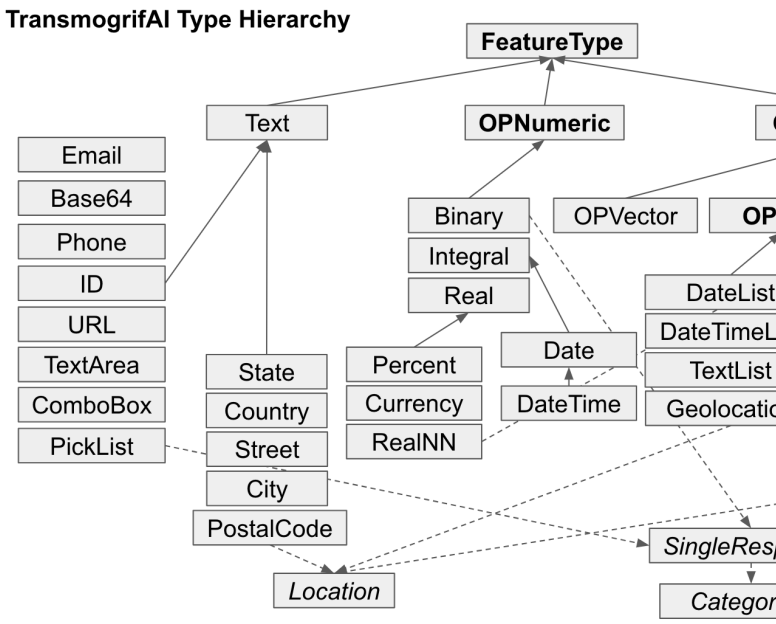
3.8.1 Features

Features are objects within TransmogrifAI which contain all information about what data is in a column and how it was created from previous data. The `name` value contained within the Feature is the same as the name of the column that will be created in the DataFrame when the data is materialized.

```
/**
 * Feature instance
 *
 * Note: can only be created using [[FeatureBuilder]].
 *
 * @param name        name of feature, represents the name of the column in the␣
↪dataframe.
 * @param isResponse  whether or not this feature is a response feature, ie dependent␣
↪variable
 * @param originStage reference to OpPipelineStage responsible for generating the␣
↪feature.
 * @param parents     references to the features that are transformed by the␣
↪originStage that produces this feature
 * @param uid         unique identifier of the feature instance
 * @param wtt         feature's value type tag
 * @tparam O feature value type
 */
case class Feature[O <: FeatureType] private[op]
(
  name: String,
  isResponse: Boolean,
  originStage: OpPipelineStage[O],
  parents: Seq[OPFeature],
  uid: String,
  distributions: Seq[FeatureDistributionLike] = Seq.empty
)(implicit val wtt: WeakTypeTag[O]) extends FeatureLike[O] {
  /* ... */
  def history: FeatureHistory // contains history of how feature was created
}
```

3.8.1.1 Type Hierarchy and Automatic Feature Engineering

The FeatureType associated with each Feature (`Feature[O <: FeatureType]`) must be part of our FeatureType hierarchy. When features are defined with a specific type (e.g. Email rather than Text), that type determines which Stages can be applied to the feature. In addition, TransmogrifAI can use this type information to automatically apply appropriate feature engineering manipulations (e.g. split Emails and pivot out the top K domains). This means that rather than specifying all manipulations necessary for each feature type the user can simply use the `.transmogrify()` method on all input features. Of course specific feature engineering is also possible and can be used in combination with automatic type specific transformations.



The FeatureType hierarchy of TransmogrifAI is shown below:

3.8.1.2 Feature Creation

Each Feature is created either using a FeatureBuilder (which is a special type of OpStage that takes in the type of the raw data and produces a single feature) or by manipulating other Features (by applying Stages) in order to make new Features. The full history of all stages used to make a feature are contained within the Feature values. In essence, Features contain the typed data transformation plan for the workflow the user creates.

3.8.2 FeatureBuilders

FeatureBuilders are used to specify how raw features must be extracted and aggregated from the source data. They are passed into the *DataReaders* during the feature generation stage in order to produce the data expected by later Stages (Estimators and Transformers) in the Workflow. FeatureBuilders specify all of the information needed to make the feature with various parts of the build command, for example:

```
val sex = FeatureBuilder.PickList[Passenger]
  .extract(d => d.sex.map(_.toString).toSet[String].toPickList).asPredictor
```

specifies the type of feature you are creating (`.PickList`), the type of the data you are extracting the feature from (`[Passenger]`), how to get the feature from that data (`.extract(d => d.sex.map(_.toString).toSet[String].toPickList)`) and whether the feature is a response or predictor (`.asPredictor`). In

addition the name of the feature is inferred from the variable name (`val sex`), the name can also be explicitly passed in if it needs to be different from the variable name (`FeatureBuilder.PickList[Passenger]("name")`). This is all the information that is required to extract a simple feature from a dataset.

If the feature creation will also involve aggregation across rows (for example if multiple days of snapshot data is being loaded) additional information is needed:

```
val fare = FeatureBuilder.Currency[Passenger].extract(_.fare.toCurrency)
  .aggregate(_ + _).window(Duration.standardDays(7)).asPredictor
```

Both the aggregation function (`.aggregate(_ + _)`) and the aggregation time window (`.window(Duration.standardDays(7))`) have default values which will be used if the user does not specify them. These methods specify how to combine rows and which timestamped rows to aggregate respectively.

3.8.3 Stages

Stages are the actually manipulations performed on Features. Notice that some of the manipulations are defined via Estimators and some via Transformers. An Estimator defines an algorithm that can be applied to one or more Features to produce a Transformer. The key difference between the two types of Stages is that Estimators can access all the information in the column or columns pointed to by the input Features. Transformers, on the other hand, simply act as transformations applied to the Feature value of a single row of the input data. The `NormalizeEstimator`, for instance, computes the mean and standard deviation of the input Feature (e.g. age) in order to produce a Transformer. Estimators are converted to Transformers when a Workflow is fitted.

ML algorithms, such as Logistic Regression, are examples of Estimators that when fitted to data produce a fitted Transformer (or Model - which is just a Transformer that was produced by an Estimator). All Estimators and Transformers are used to produce new Features which can then be manipulated in turn by any Estimator or Transformer that accepts that Feature type as an input.

3.8.4 Transformers

Once you have specified how the raw features are extracted, you can also specify how you would like them to be transformed via Transformers.

Transformers are a Spark ML concept that describes classes which perform a map operation from one or several columns in a dataset to a new column. The important thing to keep in mind about Transformers is that because they are map operations they act only within a row - no information about the contents of other rows can be integrated into transformers directly. If information about the contents of the full column are needed (for instance if you need to know the distribution of a numeric column) you must use an *Estimator* to produce a transformer which contains that information.

3.8.4.1 TransmogrifAI Transformers

TransmogrifAI Transformers extend Spark Transformers but are designed to interact with Features rather than DataFrame column names. Rather than directly passing in the data to transform function, the input Feature (or Features) are set using the setInput method and the return Feature is obtained using the getOutput method. Both the input and output Features can then be acted on as many times as desired to obtain the final result Features. When a Workflow is called, the data described by the output Feature is materialized by passing the data described by the input Features into the transformer.

3.8.4.2 Writing your own transformer

TransmogrifAI Transformers can easily be created by finding the appropriate base class and extending it. The TransmogrifAI Transformer base classes have default implementations for all the book keeping methods associated with Spark and OP. Thus when using OpTranfomer base classes the only things that need to be defined are the name of the operation associated with your Transformer, a default uid function for your transformer, and the function that maps the input to the output.

Note that when creating a new stage (Transformer or Estimator) the uid should always be a constructor argument. If the uid is not a constructor the stages will not be serialized correctly when models are saved.

TransmogrifAI Transformer base classes are defined by the number of input features and the number of output features. Transformers with a single output feature can take one (UnaryTransformers), two (BinaryTransformers), three (TernaryTransformers), four (QuaternaryTransformers), or a sequence (of a single type - SequenceTransformers) inputs. If multiple outputs are desired multistage Transformers can be used to generate up to three output features.

Lambda expressions

If the transformation you wish to perform can be written as a function with no external inputs the lambda base classes can be used.

Simply find the appropriate base class:

```
class UnaryLambdaTransformer[I, O]
(
  override val operationName: String,
  val transformFn: I => O,
  override val uid: String = UID[UnaryTransformerBase[I, O])
  ) (implicit override ...)
  extends UnaryTransformer[I, O](operationName = operationName, uid = uid)
```

And extend it to create the new transformer:

```
class LowerCaseTransformer
(
  override val uid: String = UID[LowerCaseTransformer]
) extends UnaryTransformer[Text, Text](
  operationName = "lowerCase",
  transformFn = _.map(_.toLowerCase),
  uid = uid
)
```

Creating your own params

Sometimes transformation functions could be carried out in multiple ways. Rather than creating many very similar transformers it is desirable to allow parameters to be set for transformations so that a single transformer can carry out any of these functions. The lambda transformer base classes cannot be used in this instance. Rather the class from which they inherit the *aryTransformer class should be used:

```
abstract class UnaryTransformer[I, O]
(
  override val operationName: String,
  val uid: String
)(implicit ...) extends OpTransformer1[I, O]
```

This abstract class does not take the transformFn as a constructor parameter but requires that it be implemented in the body of the new class. By having the function in the body of the class it can access parameters defined within the class. All parameters should be written using the Spark Params syntax. This is important because it marks these vals as parameters so that they can be manipulated by the workflow in addition to direct set methods.

```
class TextTokenizer
(
  override val uid = UID[TextTokenizer]
) extends UnaryTransformer[Text, PickList](
  operationName = "tokenize"
  uid = uid
)(

  final val numWords = new IntParam(
    parent = this,
    name = "numWords",
    doc = s"number of words to include in each token, eg: 1, bigram, trigram",
    isValid = ParamValidators.inRange(
      lowerBound = 0, upperBound = 4,
      lowerInclusive = false, upperInclusive = false
    )
  )

  setDefault(numWords, 1)

  def setNumWords(v: Int): this.type = set(numWords, v)

  override def transformFn: (Text) => PickList = in => {
    $(numWords) match {
       case 1 => ...
       case 2 => ...
       case 3 => ...
  }
)
```

Testing your transformer

As part of TransmogrifAI Test Kit we provide a handy base class to test transformers: `OpTransformerSpec`. It includes checks that transformer's code & params are serializable, transformer transforms data & schema as expected, as well as metadata checks. Below is example of a transformer test:

```
@RunWith(classOf[JUnitRunner])
class UnaryTransformerTest extends OpTransformerSpec[Real,␣
↪UnaryLambdaTransformer[Real, Real]] {

  /**
   * Input Dataset to transform
   */
  val (inputData, f1) = TestFeatureBuilder(Seq(Some(1), Some(2), Some(3), None).map(_.
↪toReal))

  /**
   * [[OpTransformer]] instance to be tested
   */
  val transformer = new UnaryLambdaTransformer[Real, Real](
    operationName = "unary",
    transformFn = r => r.v.map(_ * 2.0).toReal
  ).setInput(f1)

  /**
   * Expected result of the transformer applied on the Input Dataset
   */
```

(continues on next page)

(continued from previous page)

```
  val expectedResult = Seq(Real(2), Real(4), Real(6), Real.empty)

  // Any additional tests you might have
  it should "do another thing" in {
    // your asserts here
  }
}
```

3.8.4.3 Wrapping a SparkML transformer

Many of SparkML's transformers inherit from their UnaryTransformer, which is an abstract class for transformers that take one input column, apply a transformation, and output the result as a new column. An example of such a transformer is Normalizer, which normalizes a vector to have unit norm using the given p-norm. We can use the Normalizer to illustrate how to wrap a SparkML transformer that inherits from their UnaryTransformer:

```
val sparkNormalizer = new Normalizer().setP(1.0)

val wrappedNormalizer: OpUnaryTransformerWrapper[OpVector, OpVector, Normalizer] =
    new OpUnaryTransformerWrapper[OpVector, OpVector, Normalizer](sparkNormalizer)

val normalizedFeature: Feature[OpVector] = wrappedNormalizer
   .setInput(unNormalizedfeature).getOutput()
```

The flow illustrated above instantiates and configures the transformer we aim to wrap, Normalizer, and then to passes it in as a constructor parameter to an TransmogrifAI transformer wrapper, in this case, OpUnaryTransformerWrapper.

The spark wrappers built into TransmogrifAI allow users to take advantage of any estimator or transformer in Spark ML, whether or not it has been explicitly wrapped for use within TransmogrifAI. We provide a set of base classes to deal with various classes of spark stages to make this easier.

3.8.4.4 Wrapping a non serializable external library

Sometimes there is a need to use an external library from within a transformer / estimator, but when trying to do so one gets a `Task not serializable: java.io.NotSerializableException` exception. That is quite common, especially when one or more of class instances you are trying to use are not `Serializable`. Luckily there is a simple trick to overcome this issue using a singleton and a function.

Let's assume we have a text processing library, such as Lucene, we would like to use to tokenize the text in our transformer, but it's `StandardAnalyzer` is not `Serializable`.

```
import org.apache.lucene.analysis.Analyzer
import org.apache.lucene.analysis.standard.StandardAnalyzer

// We create a object to hold the instance of the analyzer
// and an apply function to retrieve it
object MyTextAnalyzer {
  private val instance = new StandardAnalyzer
  def apply(): Analyzer = instance
}

// And then instead of using the analyzer instance directly
// we refer to a function in our transformer
class TextTokenizer[T <: Text]
(
```

(continues on next page)

(continued from previous page)

```
  val analyzer: () => Analyzer = MyTextAnalyzer.apply
  uid: String = UID[TextTokenizer[_]]
)(implicit tti: TypeTag[T])
  extends UnaryTransformer[T, TextList](operationName = "txtToken", uid = uid) {

  override def transformFn: T => TextList = text => {
     // Now we safely retrieve the analyzer instance
     val tokenStream = analyzer().tokenStream(null, text)
     // process tokens etc.
  }

}
```

Note: as with any singleton object one should take care of thread safety. In the above example the `StandardAnalyzer` is thread safe, though there are scenarios where an additional coordination would be required.

Alternative solution is to create an instance of the desirable non serializable class on the fly (inside the `transformFn`). But only do so if its instance creation & destruction is lightweight, because `transformFn` is being called on per row basis.

3.8.5 Estimators

Within the context of OpWorkflows Estimators and Transformers can be used interchangeably. However, the distinction between the two classes is important to understand for TransmogrifAI developers.

Estimators are a Spark ML concept that describes classes which use information contained in a column or columns to create a Transformer which will perform a map operation on the data. The important distinction between Estimators and Transformers is that Estimators have access to all the information in the columns while transformers only act within a row. For example, if you wish to normalize a numeric column you must first find the distribution of data for that column (using the information in all the rows) using an estimator. That estimator will then produce a transformer which contains the distribution to normalize over and performs a simple map operation on each row.

3.8.5.1 TransmogrifAI Estimators

TransmogrifAI Estimators extend Spark Estimators but are designed to interact with Features rather than DataFrame column names. Rather than directly passing in the data to fit function, the input Feature (or Features) are set using the setInput method and the return Feature (or Features) are obtained using the getOutput method. Both the input and output Features can then be acted on as many times as desired to obtain the final result Features. When a Workflow is called, the data described by the output Feature is materialized by passing the data described by the input Features into the fit function and obtaining a Transformer which is then applied to the input data.

3.8.5.2 Writing your own estimator

Like TransmogrifAI Transformers, TransmogrifAI Estimators can be defined by extending appropriate base classes. Again only the name of the operation and `fitFn,` the estimation function, need to be defined.

The possible base classes are, `UnaryEstimator` (one input), `BinaryEstimator` (two inputs), `TernaryEstimator` (three inputs), `QuaternaryEstimator` (four inputs) and `SequenceEstimator` (multiple inputs of the same feature type). MultiStage Estimators can be used to return up to three outputs.

Note: that when creating a new stage (Transformer or Estimator) the uid should always be a constructor argument.** If the uid is not a constructor the stages will not be serialized correctly when models are saved.

Creating your own params

For many Estimators it is useful to be able to parameterize the way the fit function is performed. For example in writing a LogisticRegression Estimator one may wish to either fit the intercept or not. Such changes in the fit operation can be achieved by adding parameters to the Estimator. The Estimator will need to inherit from the *aryEstimator class and the companion model object will inherit from the *aryModel class. For instance, for a single input estimator the classes below should be extended:

```
abstract class UnaryEstimator[I, O]
(
  val operationName: String,
  val uid: String
)(implicit ...) extends Estimator[UnaryModel[I, O]] with OpPipelineStage1[I, O]


abstract class UnaryModel[I, O]
(
  val operationName: String,
  val uid: String
)(implicit ...) extends Model[UnaryModel[I, O]] with OpTransformer1[I, O]
```

The `fitFn` cannot be a constructor parameter as `transformFn` was in LambdaTransformers, rather it is created in the body of the class. By having `fitFn` implemented in the body of the class it can access parameters defined within the class. All parameters should be written using the Spark Params syntax. Using the Spark Params syntax allows the parameters to be set by the workflow as well as directly on the class and ensures that parameters are serialized correctly on save.

The `fitFn` must return an instance of the companion model class. Passing the values derived in the `fitFn` as constructor arguments into the companion model object allows these values to be serialized when the workflow is saved. *Note that the companion Model must take the Estimator uid as a constructor argument.*

```
class MinMaxScaler
(
  override val uid = UID[MinMaxScaler]
) extends UnaryEstimator[RealNN, RealNN](
  operationName = "min max scaler"
){
// Parameter
final val defaultScale = new DoubleParam(
    parent = this,
    name = "defaultScale",
    doc =  "default scale in case of min = max ",
    isValid = ParamValidators.gt(0.0)
 )
setDefault(defaultScale, 2.0)
def setDefaultScale(value: RealNN): this.type = set(defaultScale, value)

def fitFn(dataset: Dataset[I#Value]): UnaryModel[I, O] = {
        val grouped = data.groupBy()
        val maxData = grouped.max().first().getAs[Double](0)
        val minData = grouped.min().first().getAs[Double](0)
        val scale = if(minData == maxData) $(defaultScale) else maxData - minData
        new MinMaxModel(uid, minData, scale)
      }
}

class MinMaxModel(override val uid, val min: Double, val scale: Double)
 extends UnaryModel[RealNN, RealNN](uid, "min max scaler") {
  def transformFn: RealNN => RealNN =
```

(continues on next page)

(continued from previous page)

```
    (input: RealNN) => { ((input.value - min) / scale).toRealNN }
}
```

Testing your estimator

As part of TransmogrifAI Test Kit we provide a handy base class to test estimators: `OpEstimatorSpec`. It includes checks that estimator's code & params are serializable, fitted model is of expected type and verifies that the model transforms data & schema as expected, as well as metadata checks. Below is example of an estimator test:

```
@RunWith(classOf[JUnitRunner])
class UnaryEstimatorTest extends OpEstimatorSpec[Real, UnaryModel[Real, Real],␣
↪UnaryEstimator[Real, Real]] {

  /**
   * Input Dataset to fit & transform
   */
  val (inputData, f1) = TestFeatureBuilder(Seq(1.0, 5.0, 3.0, 2.0, 6.0).toReal)

  /**
   * Estimator instance to be tested
   */
  val estimator = new MinMaxNormEstimator().setInput(f1)

  /**
   * Expected result of the transformer applied on the Input Dataset
   */
  val expectedResult = Seq(0.0, 0.8, 0.4, 0.2, 1.0).map(_.toReal)

  // Any additional tests you might have
  it should "do another thing" in {
    // your asserts here
  }
}
```

3.8.5.3 Wrapping a SparkML estimator

To wrap a SparkML estimator, we follow a similar pattern as when wrapping a SparkML transformer. SparkML estimators all inherit from Estimator, with a couple of specializations, like Predictor and BinaryEstimator. We have wrapper classes for each of those, respectively: OpEstimatorWrapper, OpPredictorWrapper, etc. For example, to wrap SparkML's PCA estimator, we proceed as follows:

```
val wrappedPCA: OpEstimatorWrapper[OPVector, OPVector, PCA, PCAModel] =
   new OpEstimatorWrapper[OPVector, OPVector, PCA, PCAModel](new PCA().setK(10))
```

Basically, we instantiate and configure the estimator to be wrapped, PCA, and pass it in as a constructor parameter to an TransmogrifAI estimator wrapper.

3.8.5.4 Creating Shortcuts for Transformers and Estimators

One of the main benefits of having type information associated with Features is the ability to extend Feature operations syntax by adding shortcuts toTransformers/Estimators. Ultimately all Feature operations should be done using such shortcuts allowing cleaner and safer syntax.

For the sake of the example, let's assume we want to compute TF/IDF for the "name" text Feature. Which requires tokenizing the text, hashing it then computing the inverted document frequency. One can do as follows:

```
// Tokenize the "name"
val tokenized: Feature[PickList] = new TextTokenizer().setInput(name).getOutput()

// Then apply hashing transformer on tokenized "name"
val htf = new HashingTF().setNumOfFeatures(numFeatures).setBinary(binary)
val hashingTransformer = new OpTransformerWrapper[PickList, Vector, HashingTF](htf)
val hashed = hashingTransformer.setInput(tokenized).getOutput()

// Compute inverse document frequency
val idf = new IDF().setMinDocFreq(minDocFreq)
val idfEstimator = new OpEstimatorWrapper[Vector, Vector, IDF, IDFModel]()
val nameIdf: Feature[Vector] = idfEstimator.setInput(hashed).getOutput()
```

One can also add shortcuts to significantly shorten the code required and make the application of stages much clearer. Let's add a few shortcuts to allow us to perform the same transformations on features using Implicit Classes.

```
// Implicit classes enrich each feature type,
// see com.salesforce.op.dsl package for more examples
implicit class RichTextFeature(val f: FeatureLike[Text]) {
    def tokenize(): FeatureLike[PickList] = new TextTokenizer()
       .setInput(f).getOutput()
}
implicit class RichPickListFeature(val f: FeatureLike[PickList]) {
   def tf(numFeatures: Int = 1 << 8, binary: Boolean = false): FeatureLike[Vector] = {
      val htf = new HashingTF().setNumFeatures(numFeatures).setBinary(binary)
      new OpTransformerWrapper[PickList, Vector, HashingTF](htf)
         .setInput(f).getOutput()
   }
   def tfidf(numFeatures: Int = 1 << 8, binary: Boolean = false,minDocFreq: Int = 0):␣
↪FeatureLike[Vector] = {
      f.tf(numFeatures = numFeatures, binary = binary).idf(minDocFreq = minDocFreq)
   }
}
implicit class RichVectorFeature(val f: FeatureLike[Vector]) {
    def idf(minDocFreq: Int = 0): FeatureLike[Vector] = {
      val idf = new IDF().setMinDocFreq(minDocFreq)
      new OpEstimatorWrapper[Vector, Vector, IDF, IDFModel](idf)
         .setInput(f).getOutput()
    }
}
```

Once we defined the above implicit classes we can use them as follows:

```
// Once you have the implicits in scope you can write simply this
val nameIdf: Feature[Vector] = name.tokenize().tf().idf()
// Or even shorter
val nameIdf: Feature[Vector] = name.tokenize().tfidf(minDocFreq = 2)
```

Standard stages within TransmogrifAI have shortcuts that can be imported using:

```
import com.salesforce.op._
```

Shortcuts can also be created for custom stages and placed in a separate namespace.

3.8.5.5 Shortcuts Naming Convention

When adding shortcuts one should follow the below conventions:

1. For Binary, Ternary etc. Transformers/Estimators one should add a concrete class **AND** a shortcut - `f1.operand(f2, f3, ..., options)`
2. For Unary and Sequences Transformers/Estimators one should add **AT LEAST** a shortcut - `f1.operand(options)` and `Seq(f1, f2, ...).operand(options)` accordingly.
3. For verbs shortcuts should be: `f.tokenize`, `f.pivot`, `f.normalize`, `f.calibrate`, etc.
4. For nouns one should prepend "to", i.e `f.toPercentiles` etc.
5. For adjectives one should prepend "is", i.e `f.isValid`, `f.isDividable` etc.

3.8.6 Customizing AutoML Stages

Each of the special AutoML Estimators we talked about previously can be be customized.

3.8.6.1 Transmogrification

Automatic feature engineering can be customized completely by manual feature engineering and manual vector combination using the VectorsCombiner Transformer (short-hand `.combine()`) if the user desires to have complete control over feature engineering.

```
val normedAge = age.fillMissingWithMean().zNormalize()
val ageGroup = age.map[PickList](_.value.map(v => if (v > 18) "adult" else "child").
↪toPickList).pivot()
val combinedFeature = Seq(normedAge, ageGroup).combine()
```

This hand engineered feature can be used at the same time as default feature engineering other features by simply passing custom engineered features into `.transmogrify()` along with the features that the default should be applied to.

```
val features = Seq(fair, sex, combinedFeature).transmogrify()
```

Any feature that is already in vector format will simply be appended to the final feature vector unchanged by transmogrify.

3.8.6.2 SanityChecker

You can override defaults in the sanity checker params like so:

```
// Add sanity checker estimator

val checkedFeatures = new SanityChecker()
      .setMaxCorrelation(0.99)
      .setMinVariance(0.00001)
      .setCheckSample(1.0)
      .setRemoveBadFeatures(true)
      .setInput(label, features) // survived: response, passengerFeatures:␣
↪transformed predictor features
      .getOutput()
```

After the Sanity Checker has been fitted, one can access the metadata which summarizes the information used for feature selection.

```
val metadata = fittedWorkflow.getOriginStageOf(checkedFeatures).getMetadata()
```

The metadata contains a summary of the sanity checker run and can be cast into a case class containing this info for easy use:

```
val summaryData = SanityCheckerSummary.fromMetadata(metadata.getSummaryMetadata())
```

The summary is composed as follows :

- "featureStatistics" (SummaryStatistics) : Descriptive statistics of the inputs (label and features) :
  - "sampleFraction" (Double) : corresponds to the parameter `checkSample` of the sanity checker. It is the downsample fraction of the data.
  - "count" (Double) : size of the dataset
  - "variance" (Array[Double]) : variance of columns
  - "mean" (Array[Double]) : mean of each column
  - "min" (Array[Double]) : minimum of each column
  - "max" (Array[Double]) : maximum of each column
  - "numNull" (Array[Double]) : number of former missing elements for each column
- "names" (Array[String]) : names of label and features columns
- "dropped" (Array[String]) : names of the feature columns dropped
- "correlationsWLabel" (Correlations) : info about valid correlation (i.e. non Nan) of features with the labels column :
  - "values" (Array[Double]) : value of valid correlations features/label
  - "featuresIn" (Array[String]) : name of feature columns with non Nan correlations
- "correlationsWLabelIsNaN" (Array[String]) : names of the features that have a correlation of Nan with label column

In order to relate the statistics summary from sanity checker to the original parent features it is best to use the `workflowModel.modelInsights(feature)` *method*. This will output all the information gathered during workflow fitting formatted so that all feature statistics are grouped by the raw parent feature.

3.8.6.3 RawFeatureFilter

RawFeatureFilter is an optional stage that would ensure that the data distribution between the training and scoring set is similar. *Workflows* have `withRawFeatureFilter(Option(trainReader), Option(scoreReader), ...)` method which enables this. When the scoring reader is specified both the `readerParams` and the `alternateReaderParams` in the OpParams passed into the Workflow need contain paths for the data. You will need to set the score data path in the `alternateReaderParams`.

If only the training reader is specified the features will be checked for fill rates and correlation of filled values with the label. When both the training and scoring readers are specified the relationship between the two data sets is additionally checked for each raw feature and features which deviate beyond the specified acceptable range will be excluded from the workflow. The exclusion criteria have defaults, however you can set optional parameters for when to exclude features.

```
// Add raw feature filter estimator
val workflow = new OpWorkflow().setResultFeatures(survived, rawPrediction, prob,␣
↪prediction)
      .withRawFeatureFilter(
        trainingReader = Option(trainReader),
```

(continues on next page)

(continued from previous page)

```
        scoringReader = Option(scoringReader),
        // optional params below:
        bins = 100,
        minFillRate = 0.001,
        maxFillDifference = 0.90,
        maxFillRatioDiff = 20.0,
        maxJSDivergence = 0.90,
        maxCorrelation = 0.95,
        correlationType = CorrelationType.Pearson,
        protectedFeatures = Array.empty[OPFeature]
      )
```

3.8.6.4 ModelSelector

It is possible to set the type of validation as well as validation parameters such as the number of folds, the evaluation metric (`AUROC` or `AUPR` etc), or the subset of model types to try:

```
val modelSelector = BinaryClassificationModelSelector
  .withCrossValidation(numFolds = 10, validationMetric = Evaluators.
↪BinaryClassification.auROC,
     modelTypesToUse = Seq(OpLogisticRegression, OpRandomForestClassifier))
  .setInput(survived, passengerFeatures)
```

Before evaluating each model, it is possible for the BinaryClassificationModelSelector to balance the dataset by oversampling the minority class and undersampling the majority class. The user can decide on the balancing by for instance setting the targeted proportion for the minority class the balanced dataset:

```
val modelSelector = BinaryClassificationModelSelector
  .withTrainValidationSplit(Option(DataBalancer(reserveTestFraction = 0.1,␣
↪sampleFraction = 0.2)))
  .setInput(survived, passengerFeatures)
```

Similarly the MultiClassificationModelSelector allows the specification of the maximum number of labels that are allowed and the minimum support for each label:

```
val modelSelector = MultiClassificationModelSelector
  .withTrainValidationSplit(Option(DataCutter(reserveTestFraction = 0.1,␣
↪maxLabelCategories = 100, minLabelFraction = 0.01)))
  .setInput(survived, passengerFeatures)
```

Finally, it is possible to directly pass in both the models and their associated hyper parameters for any model selector. The simples way to do this is to use the spark ParamGridBuilder and do a grid search:

```
val lr = new OpLinearRegression()
val lrParams = new ParamGridBuilder()
   .addGrid(lr.fitIntercept, true)
   .addGrid(lr.regParam, Array(0.001, 0.005, 0.01, 0.05, 0.1, 0.15, 0.2))
   .addGrid(lr.elasticNetParam, Array(0.0, 0.1, 0.5))
   .build()

val myModel = new MyCustomModel() // must extend BinaryEstimator[RealNN, OPVector,␣
↪Prediction]
val myParams = new ParamGridBuilder()
   .addGrid(myModel.myParam, Array(1, 2, 3))
```

(continues on next page)

(continued from previous page)

```
   .build()

val models = Seq(lr -> lrParams, myModel -> myParams)

val modelSelector = RegressionModelSelector.withCrossValidation(
   dataSplitter = Option(DataSplitter(reserveTestFraction = 0.1)),
   modelsAndParameters = models)
```

If you wished to use a fancier hyperparameter optimization you could also directly insert the ParamMaps for each model to support bayesian or random hyperparameter search.

The return type of the ModelSelectors is a Prediction feature. This is a map type feature which contains the prediction of the estimator. For models that produce raw prediction and normalized raw prediction (or probability) values, eg. classification models, these values are also contained within the map.

```
val pred: Feature[Prediction] =  modelSelector.getOutput()
```

3.8.7 Interoperability with SparkML

All TransmogrifAI Stages can be used as spark ML stages by passing a Dataset or DataFrame directly into the `.fit()` or `.transform()` method. The important thing to note when using stages in this manner is that the **names of the input features for the Stage must match the names of the column** you wish to act on.

3.8.8 Workflows

Workflows are used to control the execution of the ML pipeline once the final features have been defined. Each Feature contains the history of how it is defined by tracking both the parent Features and the parent Stages. However, this is simply a *description* of how the raw data will be transformed in order to create the final data desired, until the Features are put into a Workflow there is no actual data associated with the pipeline. OpWorkflows create and transform the raw data needed to compute Features fed into them. In addition they optimize the application of Stages needed to create the final Features ensuring optimal computations within the full pipeline DAG. OpWorkflows can be fit to a given dataset using the `.train()` method. This produces and OpWorkflowModel which can then be saved to disk and applied to another dataset.

3.8.8.1 Creating A Workflow

In order to create a Workflow that can be used to generate the desired features the result Features and the Reader (or data source) must be defined.

```
// Workflow definition with a reader (readers are used to manipulate data before the␣
↪pipeline)
val trainDataReader = DataReaders.Simple.avro[Passenger](key = _.passengerId)

val workflow = new OpWorkflow()
   .setResultFeatures(prediction)
   .setReader(trainDataReader)

// Workflow definition by passing data in directly
val workflow = new OpWorkflow()
   .setResultFeatures(prediction)
   .setInputDataSet[Passenger](passengerDataSet) // passengerDataSet is a␣
↪DataSet[Passenger] or RDD[Passenger]
```

DataReaders are used to load and process data before entry into the workflow, for example aggregation of data or joining of multiple data sources can easily be performed using DataReaders as described in the *DataReaders* section below. If you have a dataset already loaded and simply wish to pass it into the Workflow the `setInputDataSet` and `setInputRdd` methods will create a simple DataReader for you to allow this.

It is important to understand that up until this point nothing has happened. While all the Features, Stages (transformers + estimators), and data source have been defined, none of the actual data associated with the features has been computed. Computation does not happen and Features are not materialized until the Workflow is fitted.

3.8.8.2 Fitting a Workflow

When a workflow gets fitted

```
val model: OpWorkflowModel = workflow.train()
```

a number of things happen: the data is read using the DataReader, raw Features are built, each Stage is executed in sequence and all Features are materialized and added to the underlying Dataframe. During Stage execution, each Estimator gets fitted and becomes a Transformer. A fitted Workflow (eg. a WorkflowModel) therefore contains sequence of Transformers (map operations) which can be applied to any input data of the appropriate type.

3.8.8.3 Fitted Workflows

A WorkflowModel, much like a Spark Pipeline, can be used to score data ingested from any DataReader that returns the appropriate data input type. If the score method is called immediately after train, it will score the data used to train the model. Updating the DataReader_ _allows scoring on a test set or on production data.

```
val trainDataReader = DataReaders.Aggregate.avro[Passenger](key = _.passengerId, path␣
↪= Some("my/train/data/path")) // aggregates data across many records
val workflowModel = new OpWorkflow()
   .setResultFeatures(prediction)
   .setReader(trainDataReader)
   .train()
val scoredTrainData = workflowModel.score()

val testDataReader = DataReaders.Simple.avro[Passenger](key = _.passengerId, path =␣
↪Some("my/test/data/path")) // simply reads and returns data
val scoredTestData = workflowModel.setReader(testDataReader).score()
```

The `score` method will execute the entire pipeline up to the final result Feature and return a DataFrame containing the result features.

It is also possible to compute and return the DataFrame up to any intermediate feature (for debugging workflows). This call takes the Feature that you wish to compute up to as an input and returns a DataFrame containing all raw and intermediate Features computed up to the level of the DAG of that Feature (note that the result features and reader must be set on the workflow in order to define the DA.

```
// same method will work on workflows and workflowModels
val df = workflow.computeDataUpTo(normedAge)
val df = workflowModel.computeDataUpTo(normedAge)
```

Here it is important to realize that when models are shared across applications, or namespaces, all Features must be within the scope of the calling application. A user developing a model with the intention of sharing it must take care to properly define and expose all Features.

3.8.8.4 Saving Workflows

Fitted workflows can be saved to a file which can be reloaded and used to score data at a later time. The typical use case is for one run to train a workflow, while another run will use this fitted workflow for scoring. Saving a workflow requires the use of dedicated save method.

```
val workflowModel: OpWorkflowModel = workflow.train()
workflowModel.save(path = "/my/model/location", overwrite = true)
```

The saved workflow model consists of several json files that contain all the information required to reconstruct the model for later scoring and evaluation runs. We store the computed metadata, information on features, transformers and fitted estimators with their Spark parameters.

For features we store their type, name, response or predictor, origin stage UID and parent feature names.

Persisting transfomer stages is trivial. We save transformer's class name, UID and Spark param values provided during training.

Estimators are a bit trickier. We save the class name of the estimator model, UID, constructor arguments and Spark param values provided during training. Using the class name and the constructor arguments we then able to reconctruct the instance of the estimator model and set the Spark param values.

3.8.8.5 Loading saved Workflows

Just like saving a workflow, loading them requires the use of dedicated load method on the workflow. Note that you have to use the exact same workflow that was used during training, otherwise expect errors.

```
// this is the workflow instance we trained before
val workflow = ...
// load the model for the previosely trained workflow
val workflowModel: OpWorkflowModel = workflow.loadModel(path = "/my/model/location")
```

When loading a model we match the saved features and stages to the ones provided in the workflow instance using their UIDs. For each transformer we simply set its Spark params. The estimator models are contructred using reflection using their class names and costructor arguments, and finally their Spark params are set. The operation of assigning UIDs is based on a process global state, therefore loading models is not a thread safe operation. Meaning if you are going to load models in a multi-threaded program make sure to synchronize access to `loadModel` call accordingly.

Once loaded, the model can be modified and operated on normally. It is important to note that readers are not saved with the rest of the OpWorkflowModel, so in order to use a loaded model for scoring a DataReader must be assigned to the OpWorkflowModel.

```
val results = workflowModel
  .setReader(scoringReader) // set the reader
  .setParameters(opParams)  // set the OpParams
  .score()                  // run scoring or evaluation
```

3.8.8.6 Removing problematic features

One of the fundamental assumptions of machine learning is that the data you are using to train your model reflects the data that you wish to score. TransmogrifAI has an optional stage after data reading that allows to to check that your features do not violate this assumption and remove any features that do. This stage is called the RawFeatureFilter, and to use it you simply call the method `withRawFeatureFilter(Some(trainReader), Some(scoreReader),...)` on your Workflow. This method takes the training and scoring data readers as well as some optional settings for when to exclude features (If you specify the data path in the OpParams passed into the

Workflow you will need to set the score data path in the `alternateReaderParams`). It will load the training and scoring data and exclude individual features based on fill rate, retaliative fill rate between training ans scoring, or differences in the distribution of data between training and scoring. Features that are excluded based on these criteria will be blacklisted from the model and removed from training.

This stage can eliminate many issues, such as leakage of information that is only filled out after the label and changes in data collection practices, before they effect your model. In addition because this is applied immediately after feature extraction it can greatly improve performance if there are many poor features int the data. Addition issues with features can be detected by the SanityChecker, however these checks occur on features that have undergone feature engineering steps and so often detect different kinds of data issues.

3.8.8.7 Extracting ModelInsights from a Fitted Workflow

Once you have fit your Workflow you often wish to examine the results of the fit to evaluate whether your should use the workflow going forward. We provide two mechanisms for examining the results of the workflow.

The first is the `.summary()` (or `summaryJson()`) method that pulls all *metadata* generated by the stages in the workflow into a string (or JSON) for consumption.

The second mechanism is the `modelInsights(feature)` method. This method take the feature you which to get a model history for (necessary since multiple models can be run in the same workflow) and extracts as much information as possible about the modeling process for that feature, returning a ModelInsights object. This method traces the history of the input feature (which should be the output of the model of interest) to find the last ModelSelector stage aplied and the last SanityChecker stage applied to the feature vector that went into creating the model output feature. It collects the metadata from all of the stages and the feature vector to create a summary of all of the information collected in these stages and groups the information so that all feature information can be traced to the raw input features used in modeling. It also contains the training parameters and stage parameters used in training for reference across runs.

3.8.8.8 Extracting a Particular Stage from a Fitted Workflow

Sometimes it is necessary to obtain information about how a particular stage in a workflow was fit or reuse that stage for another workflow or manipulation. The syntax to support these operations is shown below.

```
val indexedLabel: Feature[RealNN] = new OpStringIndexerNoFilter().setInput(label).
↪getOutput

// .... some manipulations that combine your features with the indexed label
// to give a finalScore ...

val fittedLeadWorkflow = new OpWorkflow()
    .setResultFeatures(finalScore)
    .setReader(myReader)
    .train()

// if you want to use just the fitted label indexer from the model or extract␣
↪information
// from that fitted stage, you can use the getOriginStageOf method on the fitted␣
↪WorkflowModel
val labelIndexer = fittedLeadWorkflow
.getOriginStageOf(indexedLabel).asInstanceOf[OpStringIndexerNoFilter] //


// if you want to create a new feature that uses the "fitted" indexer you can use it␣
↪as follows
val indexedLabel2: Feature[Numeric] = labelIndexer.setInput(newLabel).getOutput()
```

The Feature created by the Stage can be used as a handle to retrieve the specific stage from the fitted DAG. The returned Stage will be of type `OpPipelineStage` and so must be cast back into the original Stage type for full utility.

3.8.8.9 Adding new features to a fitted workflow

If you wish use stages from model that was perviously fit (for example to recalibrate the final score on a separate dataset) it is possible to add the fitted stages to new Workflow with the following syntax:

```
val fittedModel = ... // a fitted workflow containing model stages that you wish to␣
↪re-use without refitting to the new dataset
val workflow = new OpWorkflow().setResultFeatures(calibration).
↪setReader(calibrationReader)
val newWorkflow = workflow.withModelStages(fittedModel)
```

This will add the stages from the model to the Workflow replacing any estimators that have corresponding fitted models in the workflow with the fitted version. The two workflows and all their stages and features must be created in the same workspace as only directly matching stages will be replaced. When train is called on this Workflow only Estimators that did NOT appear in the previous DAG will be fit in order to create the new WorkflowModel. All stages that appeared in the original WorkflowModel will use the fit values obtained in the first fit (corresponding to the first dataset).

3.8.8.10 Metadata

Metadata is used to enhance the schema of a given feature (column in a dataset) with additional information not contained elsewhere. For example, if a given feature is of type OpVector, we still need a way to specify/query the names and history of the features used to create the columns within the vector. In addition, Metadata is also used to store information obtained during Estimator fitting which may useful later but is not part of the resulting Transformer. An example of this is the SanityChecker, which stores in the Metadata field diagnostic information related to data leakage analysis and feature statistics.

In cases where the Metadata will be frequently used, as described above, it is nice to create a case class to contain the Metadata information. Metadata itself is a spark class with an underlying structure of `Map[String, Any]` and remembering the exact structure for complex information sets is error prone. The case class can contain all of the neccessary information and contain a `.toMetadata()` function with a companion object containing a `fromMetadata(meta: Metadata)` function to convert back and forth. The Metadata itself is saved as a parameter on the stage `.setMetadata(meta: Metadata)` as well as being written into the schema of the output dataframe. This ensures that the information is retained and passed along with the created features.

In order to use the the OpVector Metadata one can then pull the Metadata off the stage or dataframe and convert it to the corresponding case class.

```
val metadata = workflowModel.getOriginStageOf(featureVector).getMetadata()
val vectorHistory = OpVectorMetadata(featureVector.name, metadata).getColumnHistory()
val dataset = workflowModel.score()
val vectorHistory2 = OpVectorMetadata(dataset.schema(featureVector.name)).
↪getColumnHistory()
```

The metadata from the SanityChecker is stored as summary Metadata under a special key to allow such information to be added to any features Metadata. It can also be cast to it's corresponding case class for ease of use. In the example below `checkedFeatures` is the feature vector output of the SanityChecker.

```
val metadata = workflowModel.getOriginStageOf(checkedFeatures).getMetadata()
val summaryData = SanityCheckerSummary.fromMetadata(metadata.getSummaryMetadata())
```

If you wish to combine the metadata from stages commonly used in modeling (ModelSelectors, SanityCheckers, Vectorizers) into a single easy(er) to reference case class we have provided a method for this in the *WorkflowModel* so that users don't need to stitch this information together for themselves.

In addition to accessing Metadata that is created by stages you may wish to add Metadata to stages of your own. For example if you created your own string indexer to map strings to integers (though we have a stage that does this), you might wish to save the mapping from idex back to string in the Metadata of the new column of integers you are creating. You would do this within the `fitFn` of the Estimator you are creating by using the `setMetadata(meta: Metadata)` method. You need a MetadataBuilder object to work with Metadata, which is essentially a wrapper around a Map of Map. For example, within an Estimator you would get a reference to a MetadataBuilder and use it as follows:

```
// get a reference to the current metadata
val preExistingMetadata = getMetadata()

// create a new metadataBuilder and seed it with the current metadata
val metaDataBuilder = new MetadataBuilder().withMetadata(preExistingMetadata)

// add a new key value pair to the metadata (key is a string,
// and value is a string array)
metaDataBuilder.putStringArray("Labels", labelMap.keys.toArray)
metaDataBuilder.putLongArray("Integers", labelMap.values.map(_.toLong).toArray) //␣
↪Metadata supports longs not ints

// package the new metadata, which includes the preExistingMetadata
// and the updates/additions
val updatedMetadata = metaDataBuilder.build()

// save the updatedMetadata to the outputMetadata parameter                    ␣
↪
setMetadata(updatedMetadata)
```

This metadata can be accessed later in various ways, for example, as part of a fitted model, by calling the model's getMetadata method:

```
val model = labelIndexer.fit(label)
val metaData = model.getMetadata()
```

We provide utility functions to simplify working with Metadata in RichMetadata. For example we have functions to add and get summary Metadata which are used in the workflow to log any information that has been saved as summary metadata.

3.8.9 DataReaders

DataReaders define how data should be loaded into the workflow. They load and process raw data to produce the Dataframe used by the workflow. DataReaders are tied to a specific data source with the type of the raw loaded data (for example the AVRO schema or a case class describing the columns in a CSV).

There are three types of DataReaders. *Simple DataReaders* just load the data and return a DataFrame with one row for each row of data read. *Aggregate DataReaders* will group the data by the entity (the thing you are scoring) key and combine values (with or without time filters) based on the aggregation function associated with each feature definition. For example aggregate readers can be used to compute features like total spend from a list of transactions. *Conditional DataReaders* are like aggregate readers but they allow an dynamic time cuttoff for each row that depends on fullfilment of a user defined condition. For example conditional readers can be used to compute features like total spend before a user becomes a member. These readers can be combined to join multiple datasources.

A constructor object provides shortcuts for defining most commonly used data readers. Defiing a data reader requires specifying the type of the data being read and the key for the data (the entity being scored).

```
val trainDataReader = DataReaders.Simple.avro[Passenger](
    key = _.passengerId
)
```

The data reader is often a good place for specifying pre-processing, e.g., filtering, that you would like applied to the data before features are extracted and transformed via the workflow. If you have specific types pre-processing steps to add to a reader this can be added by creating your own reader and overriding the read method used to load the data. The results of this read method are passed into the function that extracts the features from the input data to create the required Datafame for the workflow.

```
val trainDataReader = new DataReader[PassengerCensus](
    key = _.passengerId) {

    overrride protected def read(params: OpParams)(implicit spark: SparkSession):␣
↪Either[RDD[PassengerCensus], Dataset[PassengerCensus]]
            // method that reads Passenger data & Census data and joins
            // and filters them to produce PassengerCensus data
    }
```

3.8.9.1 Aggregate Data Readers

Aggregate data readers should be used when raw features extracted for each key need to be aggregated with respect to a particular point in time. It is easier to think about these kinds of features in the context of event data. Consider a table of events such as website visits, and imagine that we would like to extract features such as the number of times a visitor visited the site in the past 7 days. Here is the combination of FeatureBuilders and DataReaders that you would use to extract this sort of feature:

```
val numVisitsLast7Days = FeatureBuilder.Numeric[Visit]
    .extract(_ => 1)
    .aggregate(Sum)
    .daysAgo(7)
    .asPredictor

val dataReader = new AggregateDataReader[Visit](
    key = _.userId,
    aggregateParams = AggregateParams (
        timeStampFn = _.timeStamp
        cutOffTime = CutOffTime.UnixEpoch(1471046600)
    )
)
```

The timeStampFn in the aggregateParams specifies how the timestamps of the events are to be extracted, and the cutOffTime specifies the timestamp with respect to which the features are to be aggregated. All predictor features will be aggregated up until the cutOffTime, and all response features will be aggregated from the time following the cutOffTime. This kind of reader is useful for training a model to predict that an event will occur in a certain time window, for instance.

3.8.9.2 Conditional Data Readers

Sometimes, when estimating conditional probabilities over event data, features need to be aggregated with respect to the occurrence of a particular event, and the time of occurrence of this event may vary from key to key in the data set. This is when you would use a Conditional Data Reader. Continuing with our example of website visits, suppose we

are attempting to estimate the likelihood of a visitor making a purchase on the website after he makes a search, and one of the features we would like to use in the prediction is the number of times he visited the website before making the search. Here are the corresponding FeatureBuilders and Data Readers that would be needed:

```
val numVisitsLast7Days = FeatureBuilder.Numeric[Visit]
    .extract(_ => 1)
    .aggregate(Sum)
    .daysAgo(7)
    .asPredictor

val willPurchase = FeatureBuilder.Binary[Visit]
    .extract(_ => _.madePurchase)
    .aggregate(OR)
    .asResponse

val dataReader = new ConditionalDataReader[Visit](
    key = _.userId
    conditionalParams = ConditionalParams(
        timeStampFn = Option(_.timeStamp), // function for extracting
                                           // timestamp of event
        targetCondition = _.madeSearch, // function to figure out if
                                        // target event has occurred
        responseWindow = Some(Duration.standardDays(30)), // how many days after
                                                          // target event to include
                                                          // in response aggregation
        predictorWindow = Some(Duration.standardDays(30)), // how many days before
                                                           // target event to include
                                                           // in predictor aggregation
        timeStampToKeep = TimeStampToKeep.Min // if a particular key met the
                                              // condition multiple times, which
                                              // of the instances should be kept
                                              // in the training set
    )
)
```

Using this reader in a workflow will ensure that for every visitor, we extract features relative to the first time he did a search. The predictor features are aggregated from a 30 day window preceding the search, and the response features are aggregated from a 30 day window succeeding the search. Each individual feature can override this value and be aggregated based on the time span specified in the FeatureBuilder.

3.8.9.3 Joined Data Readers

Sometimes it is necessary to read data from multiple locations and combine it in order to create all the desired features. While you can always apply any data processing logic in the read method of your data reader, the preferred approach for joining data sources is to use a joined data reader:

```
val joinedDataReader = passengerDataReader.leftOuterJoin(shipInfoDataReader)
```

Joined data readers allow your raw FeatureBuilders to be defined with respect to the simpler base types rather than the complex joint types.

Inner, left outer and full outer joins are supported. Joins will by default use the keys specified in the reader to join the data sources. However, it is possible to specifiy an alternative key to join on for one of the tables, e.g. if you need to aggregate on a key other than the key you need to join on. Joins are done after feature extraction for each of the datasources.

Sometimes it is important to aggreagte feature information after the join has been performed, e.g. you aggreagte only

after an event in the first table has occured. We call this secondary aggreagtion and the most common use cases are supported by joined reasers. If a second aggregation phase is required it can be added using the JoinedReader method:

```
 def withSecondaryAggregation(timeFilter: TimeBasedFilter):␣
↪JoinedAggregateDataReader[T, U]
```

This will produce a reader that joins the data and then performs an aggregation after the join. The secondary aggregation will use the aggregators defined in the feature builders. The secondary aggreagtion will only occur on the right table unless the join keys are the primary key for both tables.

The results of a joined reader can be used for futher joins as desired:

```
  reader1.leftJoin(reader2).withSecondayAggreagtion(timeFilter).innerJoin(reader3)
```

3.8.9.4 Streaming Data Readers

Streaming Data Readers allow computation of scores with TransmogrifAI models over a stream of data. Below is an example usage using OpWorkflowRunner:

```
val streamingReader = StreamingReaders.avro[GenericRecord]()
val opParams = new OpParams().copy(
  modelLocation = Some(modelLocation) // model load location
  writeLocation = Some(scoresLocation), // scores write location
  readLocations = Map(streamingReader.typeName -> readLocation) // stream read␣
↪location
)
val runner = new OpWorkflowRunner(streamingReader = streamingReader /* all other args.
↪.. */ )

// run scoring over the stream
runner.run(OpWorkflowRunType.StreamingScore, opParams)(spark, streamingContext)
```

3.8.10 Evaluators

Like in Spark MLlib, TransmogrifAI comes with a bunch of metrics that can be used to evaluate predictions on data. TransmogrifAI Evaluators can return one or many metrics. User can also create their own evaluators.

3.8.10.1 Evaluators Factory

Evaluators are stored in a Factory that contains 3 categories :

- *Binary Classification Evaluators* : contains metrics such as AUROC, AUPR, Precision, Recall, F1-score, Error rate, True Positive, True Negative, False Positive and False Negative. `Evaluators.BinaryClassification`
- *Multi Classification Evaluators* : contains metrics like (weighted) Precision, (weighted) Recall, (weighted) F1-score and Error rate.`Evaluators.MultiClassification`
- *Regression Evaluators* : with metrics RMSE, MSE, MAE and R2. `Evaluators.Regression`

3.8.10.2 Single Evaluation

Like in Spark MLib's evaluators, TransmogrifAI evaluators have a `evaluate` method that returns one metric. As example, let's compute the AUROC of a transformed dataset `transformedData` with features `label` and `pred`:

```
val evaluator = Evaluators.BinaryClassification.auROC().setLabelCol(label).
↪setPredictionCol(red)
val metric = evaluator.evaluate(transformedData)
```

3.8.10.3 Multiple Evaluation

TransmogrifAI Evaluator can also evaluate multiple metrics using `evaluateAll`. For a Regression prediction:

```
val evaluator = Evaluators.Regression().setLabelCol(label).setPredictionCol(pred)
val metrics = evaluator.evaluateAll(transformedData)
```

`metrics` is a case class that contains the following metrics : RMSE, MSE, R2, MAE. For instance, to have access to the RMSE :

```
val rmse = metrics.RootMeanSquaredError
```

3.8.10.4 Creating a custom evaluator

Users can define their own custom evaluators too. As an example, let's create an multi classifcation evaluator that returns the cross entropy.

```
val crossEntropy = Evaluators.MultiClassification.custom(
      metricName = "cross entropy",
      isLargerBetter = false,
      evaluateFn = ds => ds.map { case (lbl, _, prob, _) => -math.log(prob.
↪toArray(lbl.toInt)) }.reduce(_ + _)
    )
val evaluator = crossEntropy.setLabelCol(lbl).setPredictionCol(prediction)
```

The field `isLargerBetter` is the same as the one in Spark's evaluators. It indicates whether the metric returned should be maximized. The method `evaluateFn` takes a Dataset with columns `(label, prediciton)`. For classification problems this will be flattened into `(label, rawPrediction, prediction, probability)` for compatibility with spark classification evaluators and easy access to different parts of the prediction feature. The evalutor returns the custom metric. Base classes `OpEvaluatorBase`, `OpBinaryClassificationEvaluatorBase`, `OpMultiClassificationEvaluatorBase` and `OpRegressionEvaluatorBase` are also available to define custom evaluators.

3.8.11 TransmogrifAI App and Runner

Workflows can be run as spark applications using the `OpAppWithRunner` (or `OpApp`). Extend the `OpAppWithRunner` base class) and define your workflow object and the training and testing data readers as part of the OpWorkflowRunner and then your workflow can be run as a spark app. Command line arguments are then used to specify the type of run you wish to do on the workflow (train, score, generate features, or evaluate).

```
// a simpler version with just a runner
object MyApp extends OpAppWithRunner {
  // build your workflow here
  def runner(opParams: OpParams): OpWorkflowRunner = new OpWorkflowRunner(/* workflow␣
↪*/)
}

// or a more customizable version
```

(continues on next page)

(continued from previous page)

```
object MyApp extends OpApp {
  /**
   * The main function to run your [[OpWorkflow]].
   * The simplest way is to create an [[OpWorkflowRunner]] and run it.
   *
   * @param runType  run type
   * @param opParams op params
   * @param spark    spark session
   */
  def run(runType: OpWorkflowRunType, opParams: OpParams)(implicit spark:␣
↪SparkSession): Unit = {
    // build your workflow here and then run it
    val result = new OpWorkflowRunner(/* workflow */).run(runType, opParams)
    // process the results here
  }
}
```

All the other pieces, such as `kryoRegistrator`, `sparkConf`, `appName`, `sparkSession`, `parseArgs` etc, are also easily customizable by overriding.

3.8.11.1 Parameter Injection Into Workflows and Workflow Runners

```
class OpParams
(
  val stageParams: Map[String, Map[String, Any]],
  val readerParams: Map[String, ReaderParams],
  val modelLocation: Option[String],
  val writeLocation: Option[String],
  val metricsLocation: Option[String],
  val metricsCompress: Option[Boolean],
  val metricsCodec: Option[String],
  val customTagName: Option[String],
  val customTagValue: Option[String],
  val logStageMetrics: Option[Boolean],
  val customParams: Map[String, Any]
)
```

In addition to defining the reader and the final feature to "materialize," OpWorkflows also allow the user to update/change the parameters for any stage within the workflow. Recall that stage state variables are stored as Spark Param key-value objects.

```
// Map of parameters to inject into stages.
// Format is Map(StageSimpleName -> Map(ParameterName -> Value)).
workflow.setParameters(new OpParams(stageParams = ("MyTopKStage" -> ("TopK" -> 10)))
```

Here we are resetting the "TopK" parameter of a stage with class name "MyTopKStage" to a value of 10. Caution must be exercised here because valid settings are not checked until runtime. This method is primarily designed to be used with CLI and Scheduler services as a means of injecting parameters values.

3.9 License

Copyright (c) 2017, Salesforce.com, Inc. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.