Types, Types, Types!

Embracing a hierarchy of types to simplify machine learning

Leah McGuire

Principal Member of Technical Staff, Salesforce Einstein

lmcguire@salesforce.com
@leahmcguire

Let’s make sure you are in the right talk

What I am going to talk about:

- What does machine learning mean at Salesforce
- Problems in machine learning for business to business (B2B) companies
- Automating machine learning and how our AutoML library (Optimus Prime) works
- The utility of having strongly typed features in AutoML
- What we have learned and what we are planning

Salesforce and Machine Learning

Salesforce

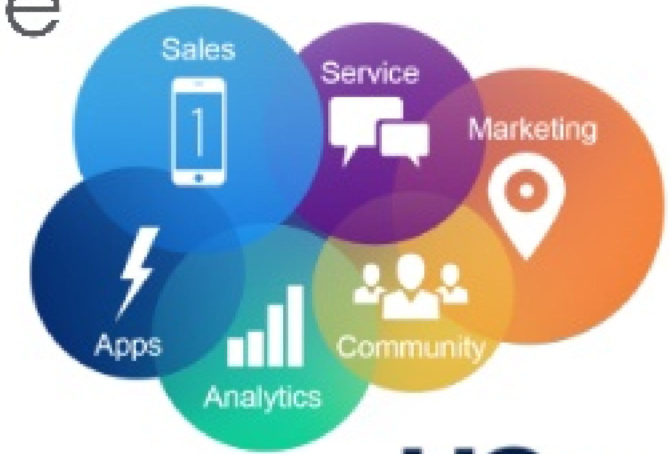
THE HOME DEPOT

Square

comcast

KOHL'S
expect great things

macy's



Uber

HORTONWORKS
POWERING THE FUTURE OF DATA

ChowNow

UCSF
University of California
San Francisco

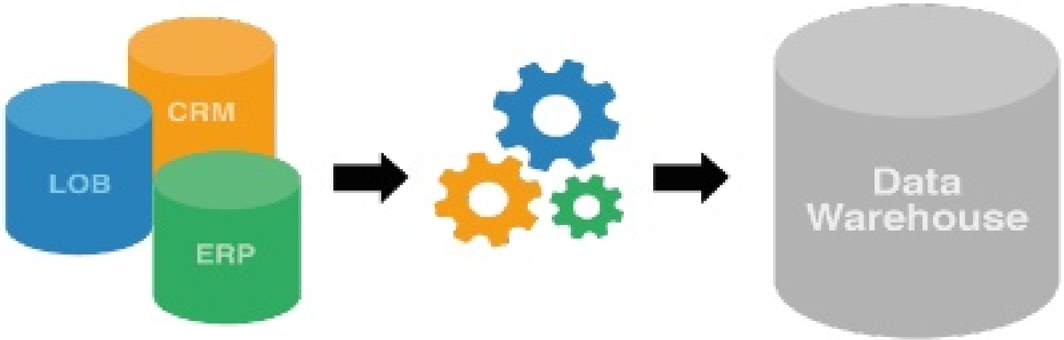
vineyard vines

FORRESTER

salesforce

The Problem

For the majority of businesses, data science is out of reach

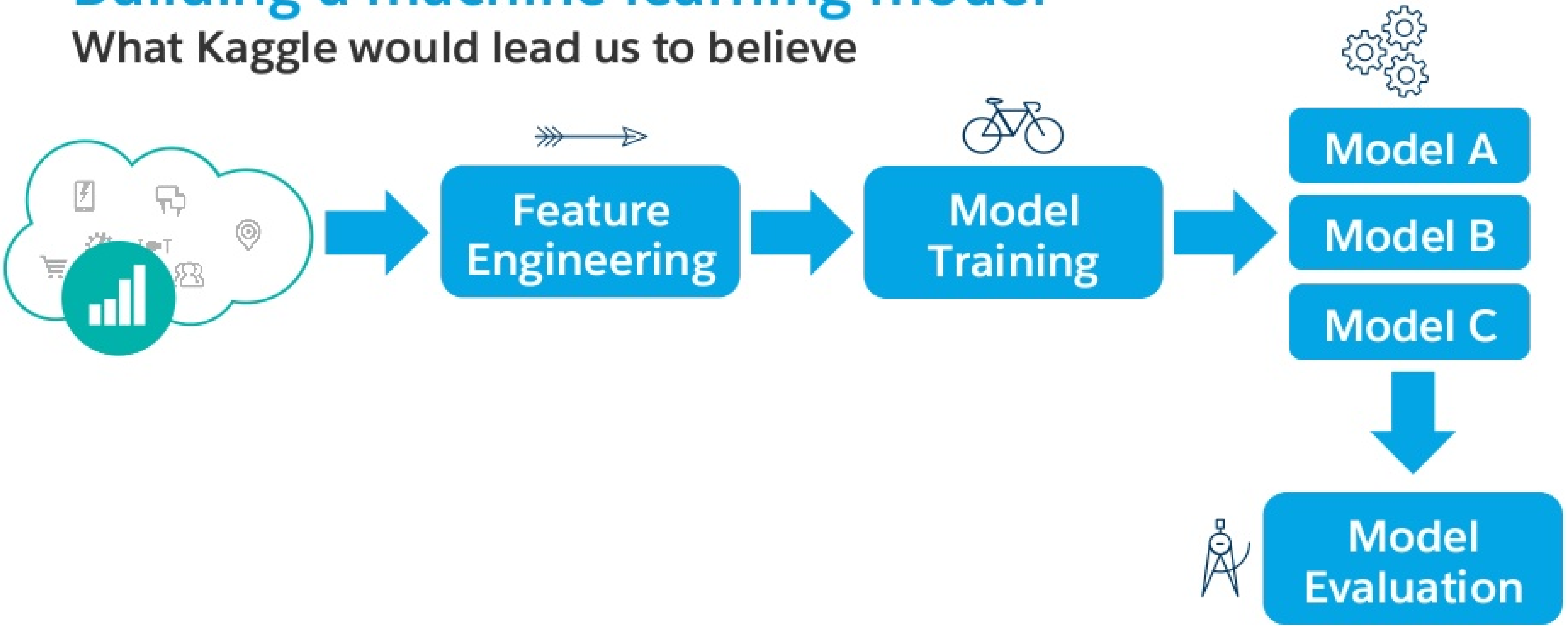


Machine learning workflows

And how much more complicated they get for B2B

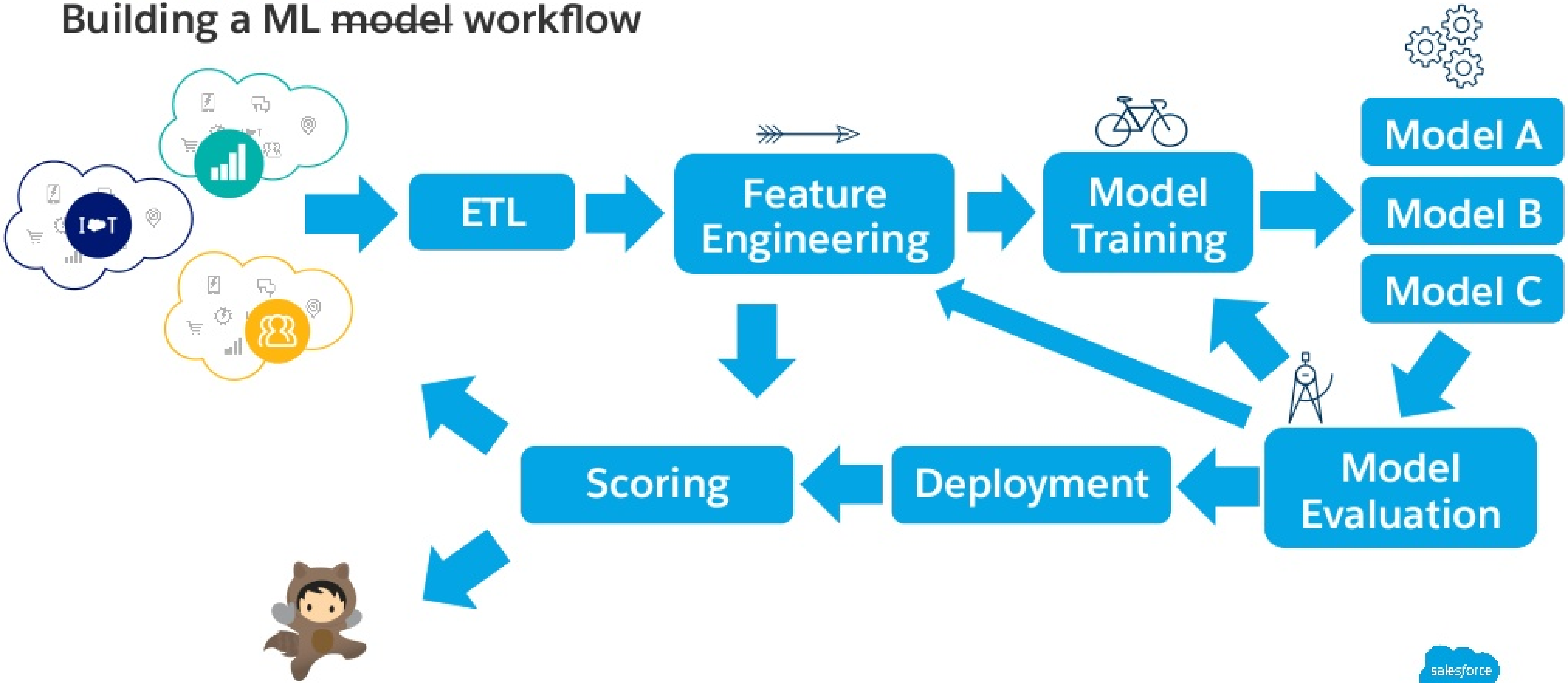
Building a machine learning model

What Kaggle would lead us to believe



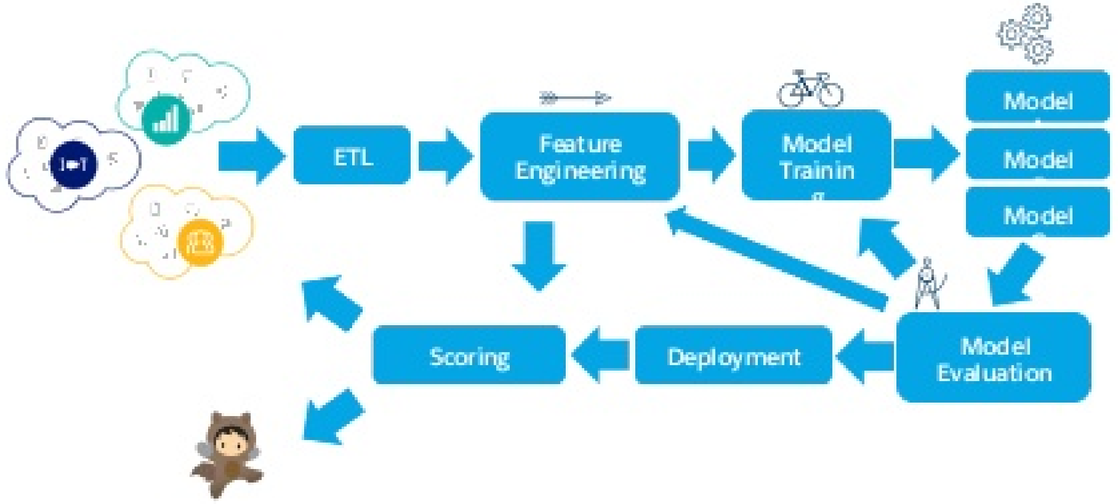
Real-life ML

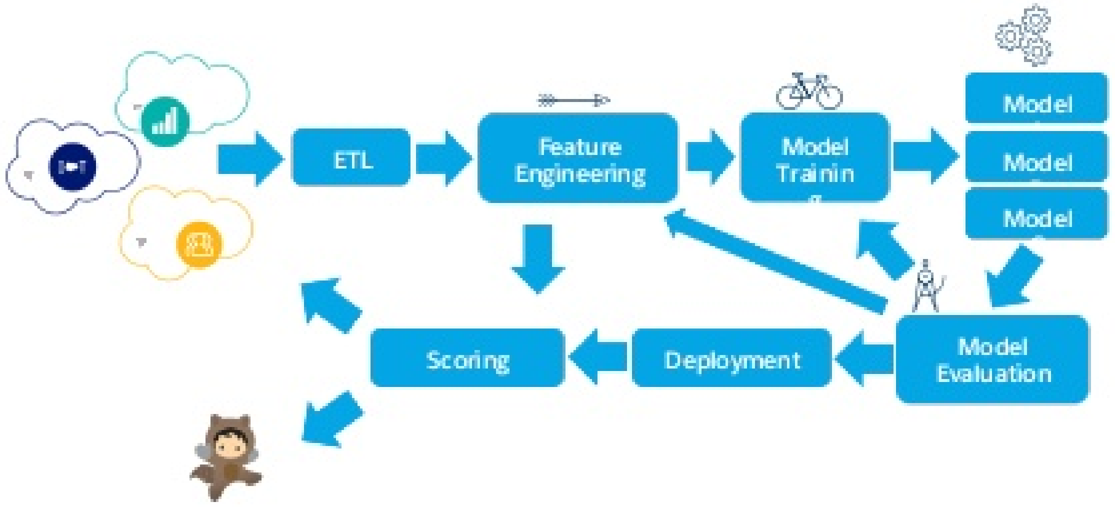
Building a ML ~~model~~ workflow

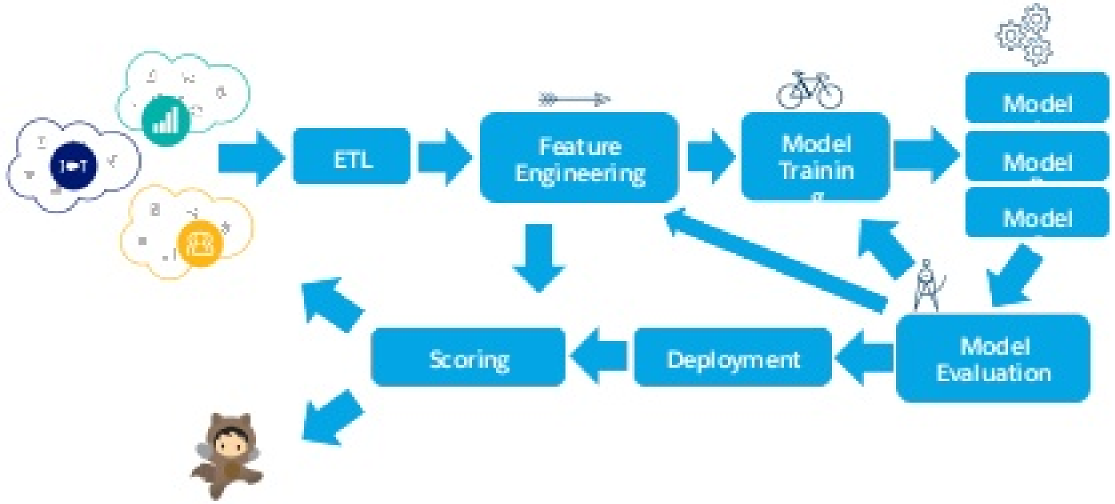


Building a machine learning model

Over and over again







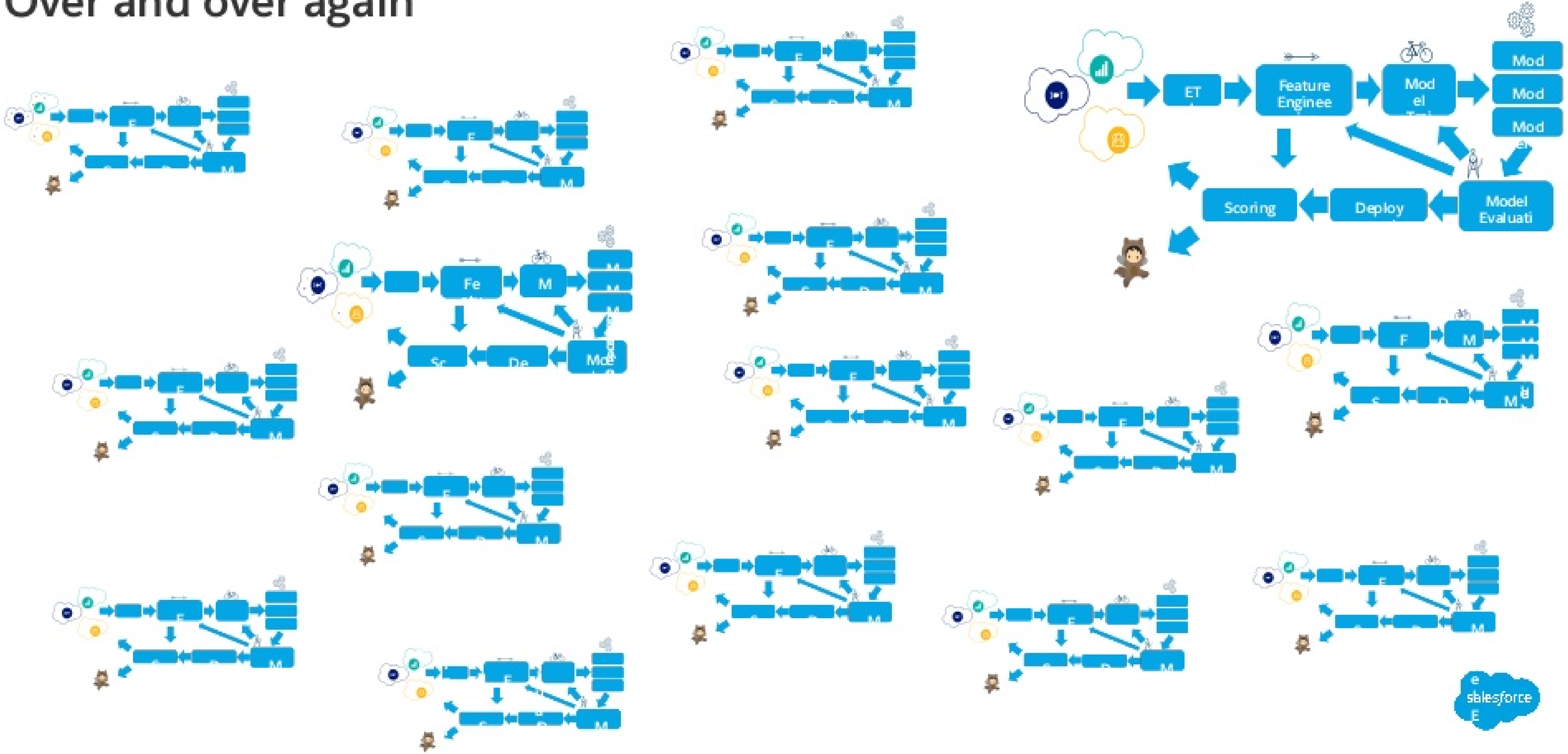
We can’t build one global model

- Privacy concerns
  - Customers don’t want data cross-pollinated
- Business Use Cases
  - Industries are very different
  - Processes are different
- Platform customization
  - Ability to create custom fields and objects
- Scale, Automation,
  - Ability to create

salesforce

Building a machine learning model

Over and over again



Automating machine learning

Enter Einstein (and Optimus Prime)

Turning a black art into a paint by number kit.

- ML is not magic, just statistics – generalizing examples
- But there is a 'black art' to producing good models
  - Input data needs to be combined, filtered, cleaned etc.
  - Producing the best features for your model takes time
  - You can't just throw a ml algorithm at your raw data and expect good results

Keep it DRY (don't repeat yourself) and DRO (don't repeat others)

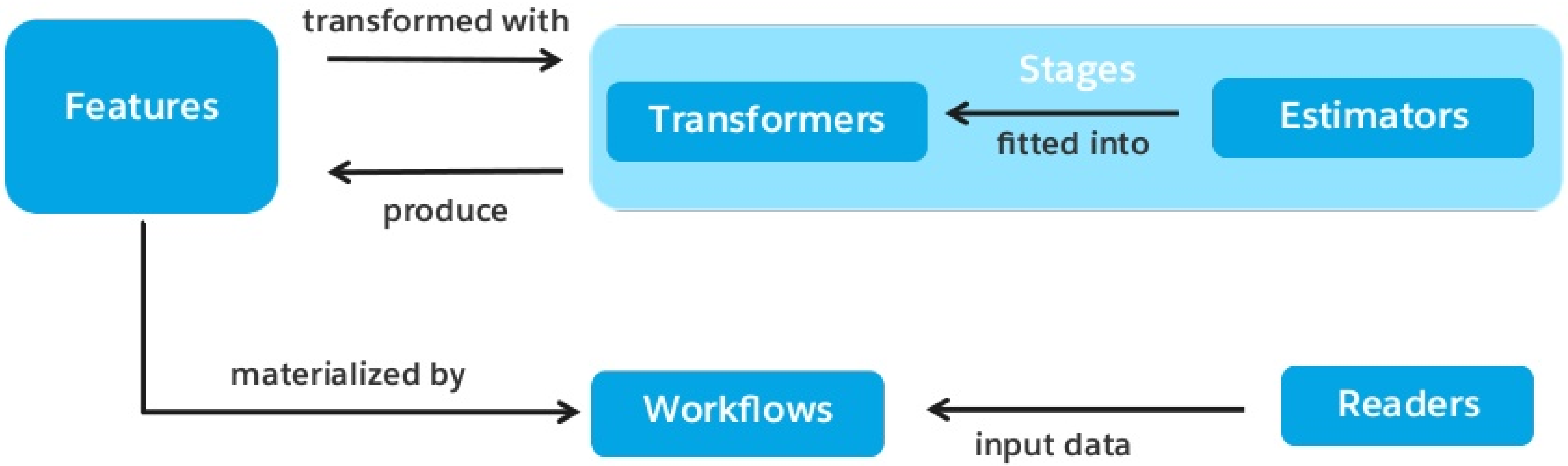
Optimus Prime - A library to develop reusable, modular and typed ML workflows

- The Spark ML pipeline (estimator, transformer) model is nice
- The lack of types in Spark is not
- Want to use more than Spark ML

- Declarative and intuitive syntax – for both workflow generation and developers
- Typed reusable operations
- Multitenant application support
- All built in scala

Simple interchangeable parts

In a declarative type safe syntax




```
val featureVector = Seq(pClass, name, gender, age, sibSp, parch, ticket, cabin, embarked).vectorize()
val (pred, raw, prob) =  featureVector.check(survived).classify(survived)
val workflow = new OpWorkflow().setResultFeatures(pred).setDataReader(titanicReader)
```

Automating typed feature engineering and modeling

(with Optimus Prime)

Features are given a type on creation

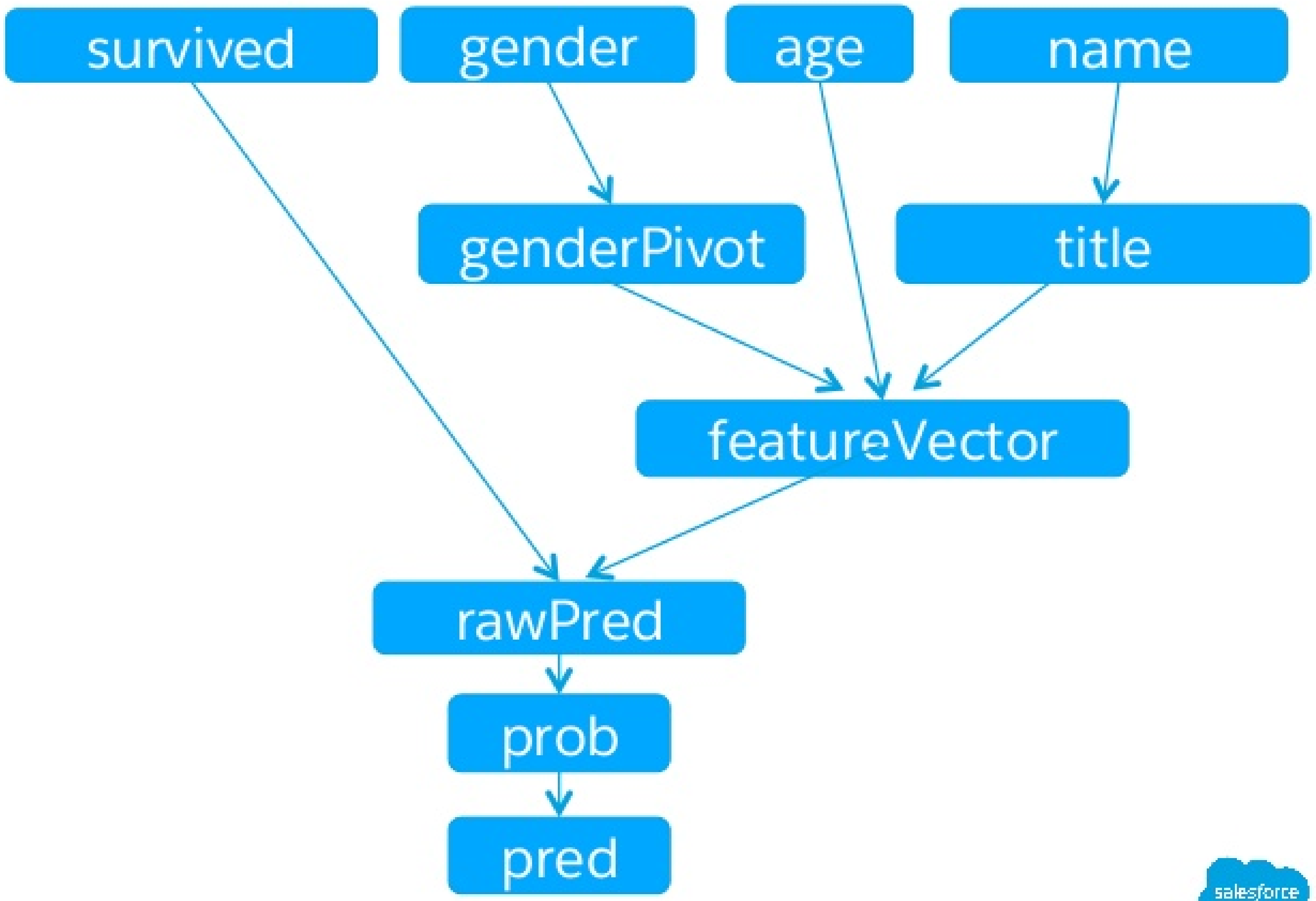
Death to runtime errors!


```
val gender = FeatureBuilder.Categorical[Titanic]
  .extract(d => Option(d.getGender).toSet[String]).asPredictor
```

- Features are strongly typed
- Each stage takes specific input type(s) and returns a specific output type(s)

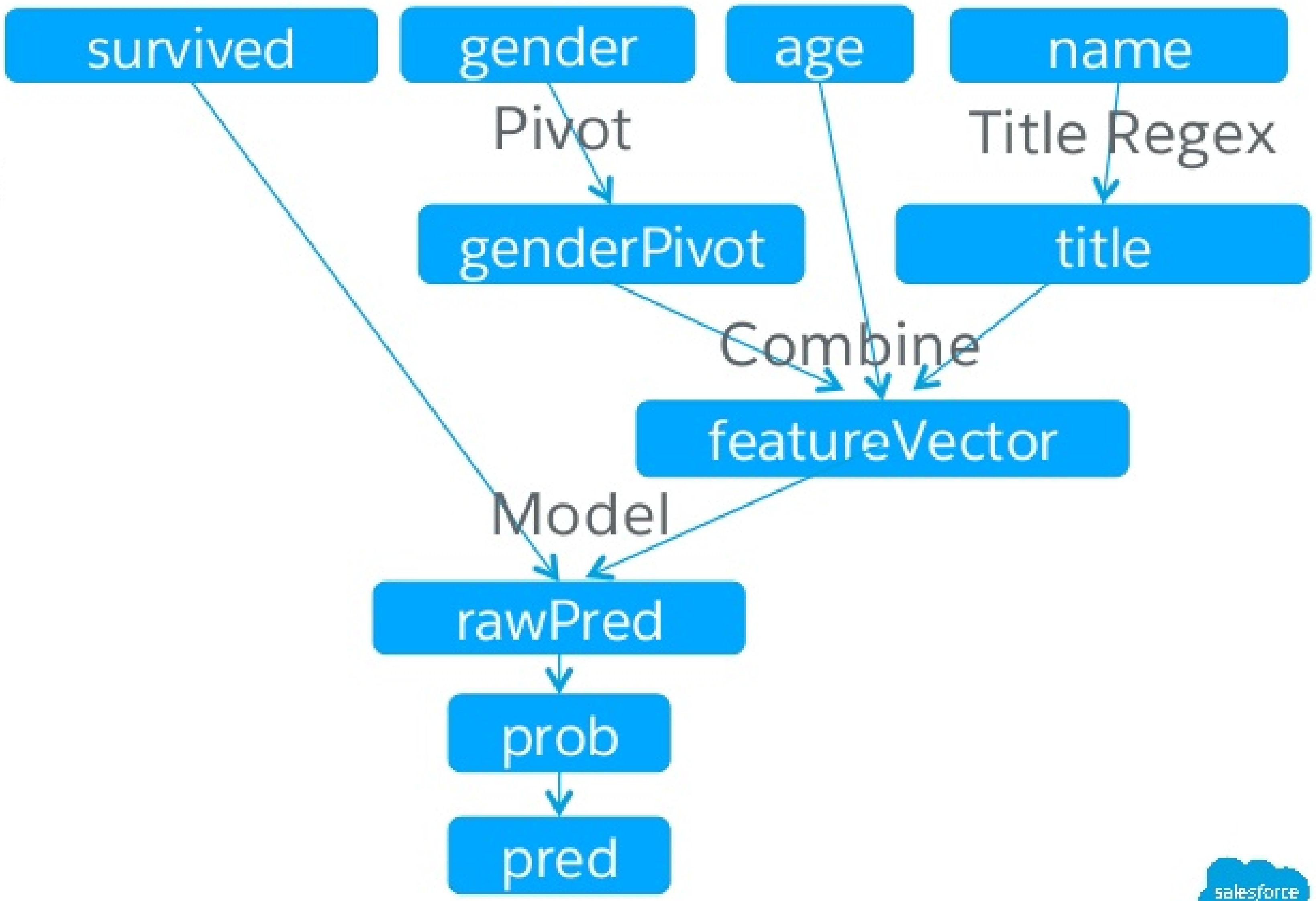
Creating a workflow DAG with features

- Features point to a column of data
- The type of the feature determines which stages can act on it

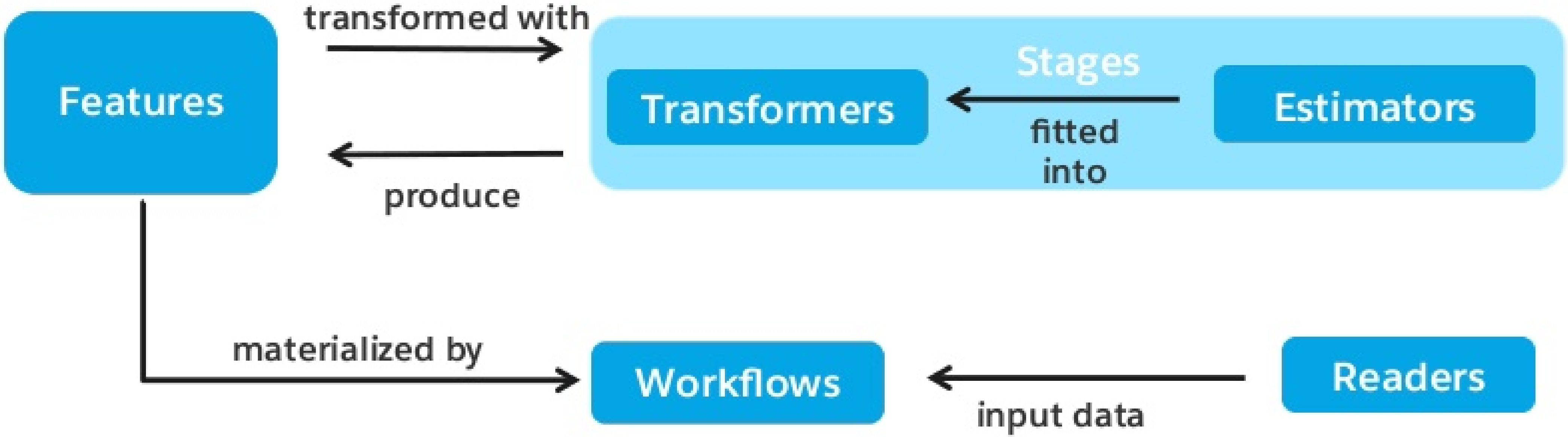


Creating a workflow DAG with features

- When a stage acts on a feature it produces a new feature (or features)
- Keep on manipulating features until you get your goal



Done manipulating your features? Make them.



- Once you make your final feature you have the full DAG
- Features are materialized by the workflow
- Initial data into the workflow provided by the reader

The power of types!

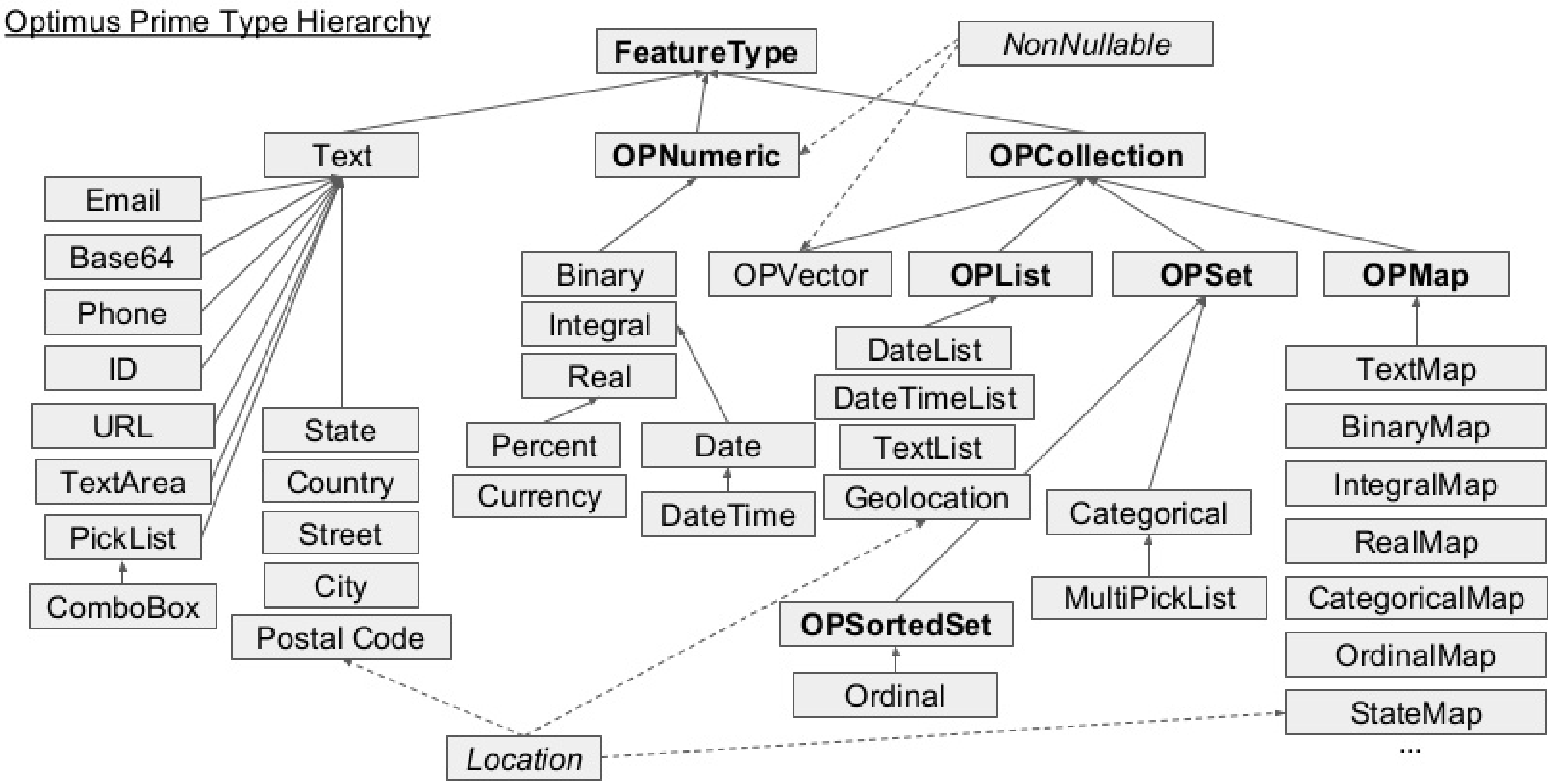
Using types to automate feature engineering

val *featureVector* = *Seq*(*pClass*, *name*, *gender*, *age*, *sibSp*, *parch*, *ticket*, *cabin*, *embarked*).vectorize()

- Each feature is mapped to an appropriate .vectorize() stage based on its type
  - *gender* (a Categorical) and *age* (a Real) are automatically assigned to different stages

- You also have an option to do the exact type safe manipulations you want
  - *age* can undergo special transformations if desired
  - val *ageBuckets* =*age*.bucketize(buckets(0, 10, 20, 40, 100))
  - val *featureVector* = *Seq*(*pClass*, *name*, *gender*, *ageBuckets*, *sibSp*, *parch*, *ticket*, *cabin*, *embarked*).vectorize()

Show me the types!



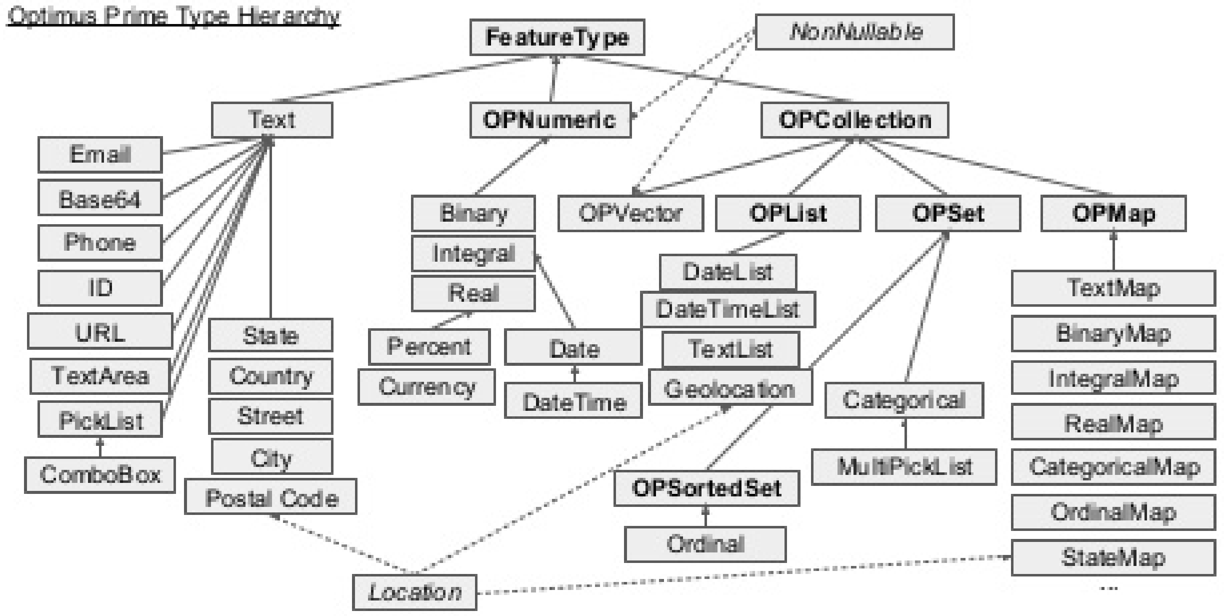
Legend: → - inheritance, **bold** - abstract class, *italic* - trait, normal - concrete class

Note: all the types are assumed to be nullable, unless NonNullable trait is mixed - https://developer.salesforce.com/docs/atlas.en-us.api.meta/api/field_types.htm

Take the types away!!

Why would we make this monstrosity??

- Sometimes a type is all you have
- Hierarchy allows both very specific and very general stages
- Type safety for production saves a lot of headaches



Legend: → - inheritance, **bold** - abstract class, *italic* - trait, normal - concrete class

Note: all the types are assumed to be nullable, unless NonNullable trait is mixed - https://developer.salesforce.com/docs/atlas.en-us.api_meta/api/field_types.htm

Sanity Checking – the stage that checks your features

- Check data quality before modeling
- Label leakage
- Features have acceptable ranges
- The feature types allow much better checks

val *checkedVector* = *featureVector*.check(*survived*)

Model Selection Stage - Resampling, Hyper-parameter Tuning, Comparing Models

- Many possible models for each class of problem
- Many hyper parameters for each type of model
- Finding the right model for THIS dataset makes a huge difference

val (*pred, raw, prob*) = *checkedFeatureVector*.classify(*survived*)

Types can save us

And if you don't believe me take a look at the code

```
val featureVector = Seq(pClass, name, gender, age, sibSp, parch, ticket, cabin, embarked).vectorize()
val (pred, raw, prob) =  featureVector.check(survived).classify(survived)
val workflow = new OpWorkflow().setResultFeatures(pred).setDataReader(titanicReader)
```

Types can save us

And if you don't believe me take a look at the code

```
val featureVector = Seq(pClass, name, gender, age, sibSp, parch, ticket, cabin, embarked).vectorize()
val (pred, raw, prob) =  featureVector.check(survived).classify(survived)
val workflow = new OpWorkflow().setResultFeatures(pred).setDataReader(titanicReader)
```

```
def addFeatures(df: DataFrame): DataFrame = {
  // Create a new family size field := siblings + spouses + parents + children + self
  val familySizeUDF = udf { (sibsp: Double, parch: Double) => sibsp + parch + 1 }

  df.withColumn("fsize", familySizeUDF(col("sibsp"), col("parch"))) // <-- full freedom to overwrite
}

def fillMissing(df: DataFrame): DataFrame = {
  // Fill missing age values with average age
  val avgAge = df.select("age").agg(avg("age")).collect.first()

  // Fill missing embarked values with default "S" (i.e Southampton)
  val embarkedUDF = udf{(e: String)=> e match { case x if x == null || x.isEmpty => "S"; case x => x}}

  df.na.fill(Map("age" -> avgAge)).withColumn("embarked", embarkedUDF(col("embarked")))
}
```

Types can save us

And if you don’t believe me take a look at the code

```
// Modify the dataframe
val allData = fillMissing(addFeatures(rawData)).cache() // <-- need to remember about caching
// Split the data and cache it
val Array(trainSet, testSet) = allData.randomSplit(Array(0.75, 0.25)).map(_.cache())

// Prepare categorical columns
val categoricalFeatures = Array("pclass", "sex", "embarked")
val stringIndexers = categoricalFeatures.map(colName =>
  new StringIndexer().setInputCol(colName).setOutputCol(colName + "_index").fit(allData)
)

// Concat all the feature into a numeric feature vector
val allFeatures = Array("age", "sibsp", "parch", "fsize") ++ stringIndexers.map(_.getOutputCol)

val vectorAssembler = new VectorAssembler().setInputCols(allFeatures).setOutputCol("feature_vector”)

// Prepare Logistic Regression estimator
val logReg = new LogisticRegression().setFeaturesCol("feature_vector").setLabelCol("survived”)

// Finally build the pipeline with the stages above
val pipeline = new Pipeline().setStages(stringIndexers ++ Array(vectorAssembler, logReg))
```

Types can save us

And if you don't believe me take a look at the code

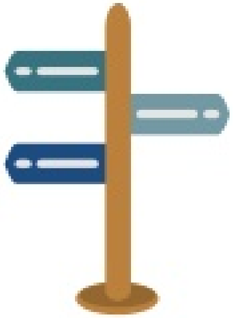
```scala
// Cross validate our pipeline with various parameters
val paramGrid =
  new ParamGridBuilder()
    .addGrid(logReg.regParam, Array(1, 0.1, 0.01))
    .addGrid(logReg.maxIter, Array(10, 50, 100))
    .build()

val crossValidator =
  new CrossValidator()
    .setEstimator(pipeline) // <-- set our pipeline here
    .setEstimatorParamMaps(paramGrid)
    .setEvaluator(new BinaryClassificationEvaluator().setLabelCol("survived"))
    .setNumFolds(3)

// Train the model & compute scores
val model: CrossValidationModel = crossValidator.fit(trainSet)
val scores: DataFrame = model.transform(testSet)

// Save the model for later use
model.save("/models/titanic-model.ml")
```

Where are we going and what have we learned

Key takeaways

- ML for B2B is a whole other beast
- Spark ML is great, but it needs type safety
- Simple and intuitive syntax saves you trouble down the road
- Types in ML are incredibly useful
- Scala has all the relevant facilities to provide the above
- Modularity and reusability is the key

Going forward with Optimus Prime



- Going beyond Spark ML for algorithms and small scale
- Making everything smarter (feature eng, sanity checking, model selection)
- Template generation
- Improvements to developer interface

If You're Curious ...

PredictionIO

HEROKU

MetaMind

einstein-recruiting@salesforce.com

Thank You