Building Smarter Application: Consumer Loan Application

Amy Wang
(+ Tom Kraljevic and
Prithvi Prabhu)

BUILDING A SMARTER APPLICATION

Q: What is a Smarter Application?
A: An application that learns from data
[From rules-based to model-based]

Topics:

- Combining applications with models
- Deploying models into production

Target audience:

- *Developers* adding Machine Learning to apps
- *Data Scientists/DevOps* putting models into production

A CONCRETE USE CASE

- We're building a consumer loan app
- The end-user is applying for a loan
- Imagine the website is a lender
- Should a loan be offered?
- Two predictive models
  - Is the loan predicted to be bad (yes/no)
  - If no, what is the interest rate to be offered?

STEPS TO BUILDING A SMARTER APP

Step 1: Picking the question your model will answer

Step 2: Using your data to build a model

Step 3: Exporting the generated model as a Java POJO

Step 4: Compiling the model

Step 5: Hosting the model in a servlet container

Step 6: Running the JavaScript app in a browser

Step 7: Using a REST API to make predictions

Step 8: Incorporating the prediction into your application

THE DATA

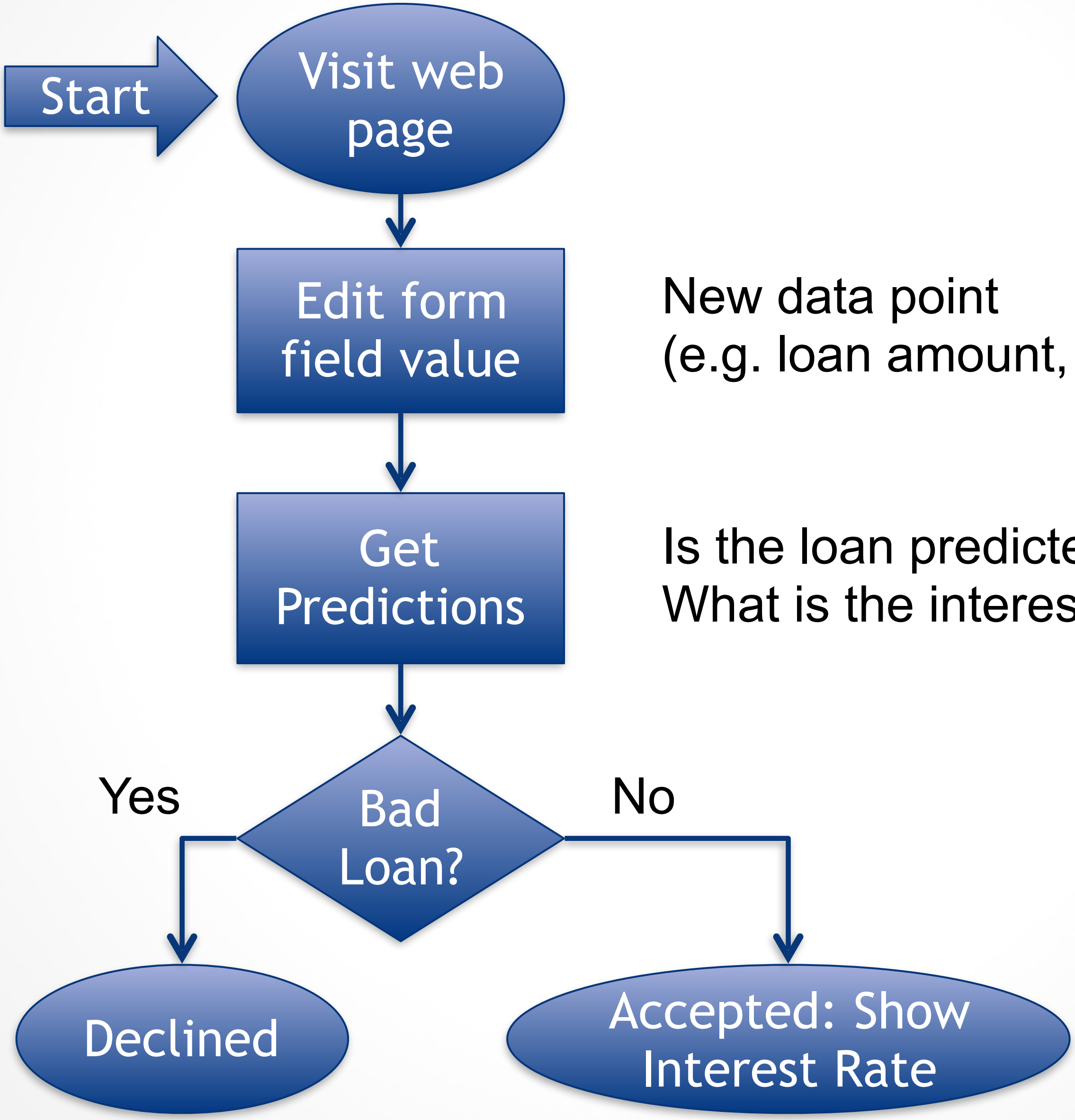
- Lending club loans from 2007 to June 2015
- Only loans that have a known good or bad outcome are used to build the model
- 163,987 rows
- 15 columns

DATA DICTIONARY

Predictor Variable	Description	Units
loan_amnt	Requested loan amount	US dollars
term	Loan term length	months
emp_length	Employment length	years
home_ownership	Housing status	categorical
annual_inc	Annual income	US dollars
verification_status	Income verification status	categorical
purpose	Purpose for the loan	categorical
addr_state	State of residence	categorical
dti	Debt to income ratio	%
delinq_2yrs	Number of delinquencies in the past 2 years	integer
revol_util	Revolving credit line utilized	%
total_acc	Total accounts (number of credit lines)	integer
longest_credit_length	Age of oldest active account	years

Response Variable	Description	Model Category
bad_loan	Is this loan likely to be bad?	Binomial classification
int_rate	What should the interest rate be?	Regression

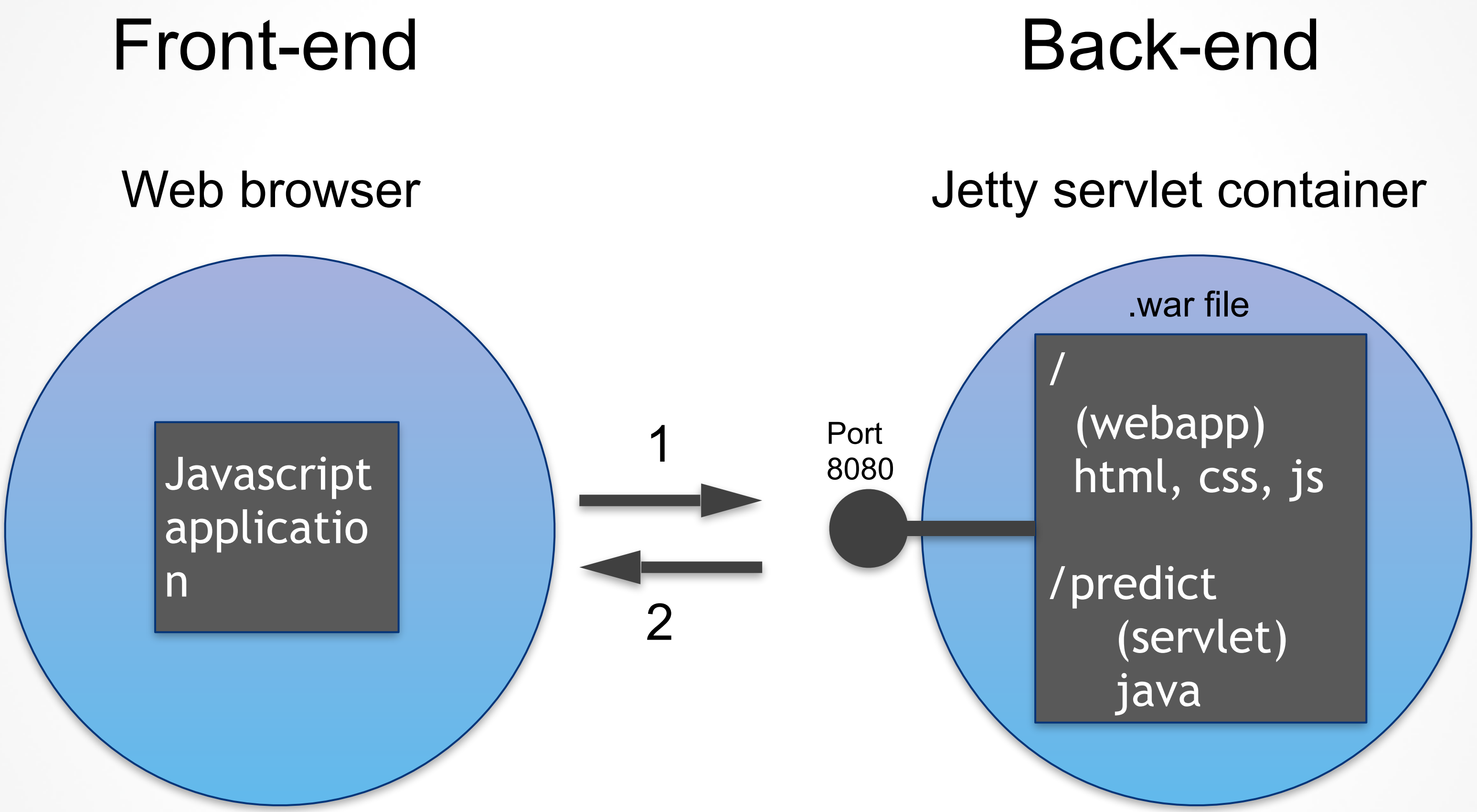
WORKFLOW FOR THIS APP



New data point
(e.g. loan amount, income)

Is the loan predicted to be bad?
What is the interest rate?

APP ARCHITECTURE DIAGRAM



1. HTTP GET with query parameters (loan_amt, annual_inc, etc.)
2. JSON response with predictions

MODEL INFORMATION

Bad Loan Model

Algorithm:	GBM
Model category:	Binary Classification
ntrees:	100
max_depth:	5
learn_rate:	0.05
AUC on valid:	.685
max F1:	0.202

Interest Rate Model

Algorithm:	GBM
Model category:	Regression
ntrees:	100
max_depth:	5
learn_rate:	0.05
MSE:	11.1
R2:	0.424

SOFTWARE PIECES

- Offline
  - R + H2O (model building)
- Online
  - Front-end
    - Web browser
    - JavaScript application (run in the browser)
  - Back-end
    - Jetty servlet container
    - H2O-generated model POJO (hosted by servlet container)

HANDS-ON DEMONSTRATION

If you are already running H2O on your laptop, please stop it so the gradle script runs properly!

STEP 1: Compile and run (From the command line)

```
./gradlew build
```

STEP 2: Use the app (In a web browser)

http://localhost:8080

STEP 3: Rerun without rebuilding the models or recompiling

```
./gradlew jettyRunWar -x war
```

COMMON HANDS-ON ERRORS

- Common R errors
  - R not on PATH
    - Gradle needs to invoke R
  - Another H2O is already running
    - the R script can’t find the data in h2o.importFile()
- Common Java errors
  - Java not installed at all
    - Also, must install a JDK (Java Development Kit) so that the Java compiler is available (JRE is not sufficient)
  - Not connected to the internet
    - Gradle needs to fetch some dependencies from the internet

KEY FILES

- Offline
  - **build.gradle**
  - **data/loan.csv**
  - **script.R**

- Front-end
  - **src/main/webapp/index.html**
  - **src/main/webapp/app.js**

- Back-end
  - **src/main/java/org/gradle/PredictServlet.java**
  - *lib/h2o-genmodel.jar (downloaded)*
  - *src/main/java/org/gradle/BadLoanModel.java (generated)*
  - *src/main/java/org/gradle/InterestRateModel.java (generated)*

POST-DEMO POINTERS

- POJO Javadoc
  - http://h2o-release.s3.amazonaws.com/h2o/rel-tibshirani/3/docs-website/h2o-genmodel/javadoc/index.html

NEXT STEPS: CLOSING THE FEEDBACK LOOP

- Scoring
  - Judging how good the predictions really are
  - Need to get the correct answers from somewhere
- Storing predictions (and the correct answers)
  - Often Hadoop
  - This can be a lot of work to organize

NEXT STEPS: RETRAINING AND DEPLOYING

- Model update frequency
  - Need depends on the use case
  - Hourly, daily, monthly?
  - Time cost of training the model is a factor

- Hot swapping the model
  - Separating front-end and back-end makes this easier
  - Java reflection for in-process hot-swap
  - Load balancer for servlet container hot-swap

RELATED EXAMPLES

- H2O Generated Model POJO in a Storm bolt
  - GitHub: h2oai/h2o-world-2015-training
  - tutorials/streaming/storm

- H2O Generated Model POJO in Spark Streaming
  - GitHub: h2oai/sparkling-water
  - examples/src/main/scala/org/apache/spark/examples/h2o/CraigslistJobTitlesStreamingApp.scala