Source Code Documentation

Master’s Thesis – Stefan Heidekrüger

The Source Code can be separated into two projects, Winning Price Prediction and Simulation of Bid Strategies. First, we will give an overview of the Source Code files by project, the classes and objects contained in each of them and their purpose.

For non-trivial classes we will explain the detailed implementations in the next in the second Section.

It should be noted that while in the source files are on disk organized into a single project “WP-Prediction”, they should logically be organized into two separate projects “WP-Prediction” and “Simulation”.

# Source Files and Classes

## Winning Price Prediction

|  |  |  |
| --- | --- | --- |
| File | Class/Object | Description |
| FeatureHasher.scala | FeatureHasher | Spark Transformer that implements Feature Hashing for categorical variables |
| FeaturePreparation.scala | FeaturePreparation | Object representing the feature preparation pipeline. In particular, it classifies input columns according to their type (categorical, categorical w/ possibly unseen values, numerical, should be hashed or not, etc.) |
| FtColumnAssembler.scala |  | Spark Transformer that combines multiple scalar features into a sparse feature vector as required by the algorithms. This is a wrapper on Spark’s VectorAssembler that automatically only selects the appropriate columns. |
| MyStringIndexer.scala | MyStringIndexer | A variation of the original StringIndexer class in Spark. Functionality for dealing with unseen features has been added, particularly considering the unseen value to be the same as “NA”. |
| Preprocessing.scala | Preprocessing | Object representing the preprocessing part of the script. Deals with input transformations and typing. Partly obsolete as was originally written for previous version of the input data. |
| Training.scala | Training | Object, is called in the main script and represents the actual model training part of the pipeline: Algorithms and their hyperparameters are set in this script. |
| WPPrediction.scala | WPPrediction extends App | Main script, performs all steps: reading in the data, preprocessing it, generating and extracting features in the required form and finally training and evaluating a model. |

## Campaign Simulation Framework

|  |  |  |
| --- | --- | --- |
| File | Classes/Objects | Description |
| BiddingStrategy.scala | BiddingStrategy | Abstract class representing a bidding strategy. Explained below |
| “ | RandomBidder, ConstantBidder, DynamicRandomBidder, LinearBidder, LinearBidderWithPacing, ValueBidder, ThresholdValueBidder | Implementations of (relatively) simple bidding strategies. Names and code/inline comments should be self-explanatory. LinearBidder, LinearBidderWithPacing implements linear-form bidding as in Section 5.2.5 in the thesis. |
| BidLandscape.scala | BidLandscape | Represents a bid landscape, i.e. the supply side of the market at a given time, particularly the rate of requests and the distribution of winning prices. This implementation assumes log-normal winning price distribution. Provides access to the winning function, summary statistics, as well as methods for mixing multiple landscapes. |
| “ | Object BidLandscapeServer | Singleton Object used to query the BidLandscapes found in the training data. In particular, this object holds a Spark-Dataframe containing the data in raw .csv format as created by TrainingMetrics.scala (see below) and efficiently serves these landscapes to the simulation either by querying the data frame (when a landscape is requested for the first time) or from memory (subsequent requests), such that Spark and filesystem operations are minimized at simulation runtime. |
| BidRequest.scala | BidRequest | Data structure that represents a bid request with all necessary data, both for simulating the auction (winning price), and as seen from a campaign’s point of view in the interim stage. *Thus ex-interim predictions for CTR and WP are included in this data structure and are not computed at runtime in this simulation framework.* |
| “ | BidResult | A simple data structure that holds the result of an auction after a bid has been submitted, such as whether the auction was won, the price paid, etc. |
| Campaign.scala | Campaign | Represents a single campaign. Explained in detail **below.** |
| “ | CampaignBidEvent | Simple data structure for representing a bid event in ac campaign. Used for for logging purposes and bookkeeping. |
| CampaignSimulation.scala | Object CampaignSimulation extends App | Main script for simulating and logging one or multiple campaigns. Explained in detail below. |
| HelperFunctions.scala | Object HelperFunctions | Provides some helper functions used throughout the project, using conversions between data structures such as events, times, etc. |
| ORTBidder.scala | ORTBidder extends BiddingStrategy | Implements ORTB Bidding as seen in Section 5.2.3 |
| RawBidRequest.scala | RawBidRequest | Simple Data Structure corresponding to a bid request as stored in the input data. |
|  | Object RawBidRequestReader | Singleton Object used in IO for converting csv-based data into Scala data structures. |
| ShadedValueBidder.scala | ShadedValueBidder extends BiddingStrategy | Implements Bid Shading Strategy as in Sections 5.2.2, 5.2.4 |

## Others (Not Needed for Testing the Code)

|  |  |
| --- | --- |
| File | Description |
| TrainingMetrics.scala | Script for finding Bid-Landscapes in Training data as described in Section 5.3.1. Resulting Bid Landscapes get saved to the file system in text format such that they can be used by BidLandscapeServer |
| FullDataBatchProcessing.scala | Script Template for processing full Training Data in its original form, was changed multiple times and used for some preprocessing and sample creation. (Obsolete) |
| CreateJoinedDataset.scala | Script Template for merging iPinYou training data with CTR-Predictions provided by Yannis. Was needed at an earlier stage in development for an interim ETL processed. (Obsolete) |

# Simulation Framework – Detailed Explanation

### Input

The simulation framework takes the following inputs, all contained in the data/simulation\_data/ folder:

* Test Set in preprocessed form, containing bid requests represented by their basic data (id, timestamp, Impression, Click, Conversion, Advertiser, Floor Price, Paying Price, Original Bid) as well as predictions on CTR and Winning Price won from the training set.
* Training Results on the bid landscape, (contained in the files training\_metrics\_hourly.csv and training\_metrics\_buckets.csv (for 6-hour buckets). While these files contain some additional information that was used for diagnostics of training, the important import data for the simulation are the identifier (Advertiser, Weekday, Hour/bucket), mean price, variance of price, number of impressions (in training set, corresponds to number of requests in simulation) and CTR. Using these data and assumptions on the form (here: lognormal), the framework can then determine the estimated distribution of winning prices.

### Main Script: CampaignSimulation.scala

This is the main script where simulation takes place. The script can run multiple campaigns at once and is generally setup such that it will perform simulations for *one specified type* of Bidding Strategy for a certain *specified range of budgets* for *all advertisers.* The strategy and budget range to be used can be set in the first few lines of the script.

The procedure is then as follows:

1. At first, the BidLandscapeServer is set up, such that campaigns can use it to query expected landscapes.
2. For each of the (#budgets x #advertisers) campaigns, a BiddingStrategy object is initiated according to the setting set in the header.
3. An Array of Campaign objects is initialized with the corresponding parameters and bidders.
4. The script traverses the input files of the test set and serves each of the bid request to each of the campaigns. The campaigns then handle each request according to logic contained in the Campaign class (see below)
5. When the run through the data is completed, the current state of each campaign is read and written to logs.

#### Logging

The script performs the following types of logging:

* Human-readible campaign logs of all campaigns in this run will be written to the output directory in a CampaignMetrics.txt file, this includes aggregate campaign level data such as spending, avg. CTR, number of clicks achieved and so on.
* The same data will be written as one row per campaign to the AllSimulations.csv file in the main log directory. This file contains all campaign-level logging data of previous runs and can be used for comparisons and analysis.
* Optionally, event-level logging can be enabled in the header of the script. This will write detailed bidding logs for each request (i.e. bid amount, lost/won, paying price, click, conversion, etc). This detailed level of logging can be useful to look into campaigns behavior over time in detail, but is very resource intensive (log for each campaign ~400MB, in current implementation this amount of RAM per campaign is also needed, as the logs are only written in the end and stored in memory while running. This can certainly be optimized if required.) and thus disabled by default and should only be used when running very few campaigns at the same time.

### The Campaign Class

Each campaign is represented by a Campaign Object, which stores the state of the campaign and handles incoming bid requests, the basic outline will be explained here, for details see the source code.

The *state* of the campaign most importantly includes the following:

* Advertiser, (*influences which ads are considered relevant)*
* BiddingStrategy (*object corresponding to the strategy. The strategy object is never changed during a campaign, may however be tuned according to current campaign state for some strategies)*
* Budget
* Time and Activity (*the campaign knows the current time and determines whether it is currently active depending on time and budget. An inactive campaign will ignore incoming bid requests. In the current set-up, the campaign will update its bid landscape and tune its strategy once every hour)*
* Current BidLandscape (*the campaign holds information about the current expected bid landscape)*
* Logging and Historical data (*this is collected for later logging as well as a possible prior on future bid landscapes. In particular, the campaign stores the following info:*
* *Aggregate campaign wide metrics, such as budget spent, impressions and clicks achieved etc.*
* Historical Bid-Landscapes (both campaign wide and in the running hour). The running hour bid landscape may be chosen as a prior for the bid landscape that is used in the next hour. This can be achieved by setting the field landscapeTrainingWeight. A value of 1.0 will always use the training landscape, 0.0 will use the landscape of the previous hour, values between 0 and 1 will use a weighted mixture of those two landscapes.  
  Historical Landscapes are updated with fixed memory requirement and without preserving event-level information using an online algorithm ([*https://en.wikipedia.org/wiki/Algorithms\_for\_calculating\_variance#Online\_algorithm*](https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance#Online_algorithm) *)*
* *If enabled, event-level bidding logs.*

While most of the class deals with handling, retrieving and updating state of the campaign, the core logic of the class is the handleBidRequest method. This method takes an incoming bid-request and deals with it according to the current campaign state. A high-level overview is given here (basic bookkeeping omitted):

1. Check validity of the requests (especially: time consistency)
2. Check whether the campaign is active, otherwise ignore any requests
3. Determine what amount to bid (if any) using the campaign’s bidding strategy
4. Bid on the request
5. If won, update state of campaign and historical bid landscapes

#### The BiddingStrategy Class

This is an abstract class representing a bidding strategy and providing an API for campaigns. Each strategy must implement the following member

* A method getBid(campaign, bidRequest) which gets the appropriate bid amount for a bidRequest according to the state of the campaign.

The following members are provided in a generic form and can be overwritten by specific strategies:

* A method tune which is used to update strategy-specific parameters dynamically. The abstract method is empty; each dynamic strategy must thus include its own update rules.
* A method getExpectedSpending which is used to determine the expected future spending of the strategy over the rest of the campaign (using its current parameters, where applicable). The generic version takes an expected average bid level and then determines the spending by using all future bid-landscapes and their respective winning functions, i.e. the expected spending for each bid-landscape-window is given by

#### The BidLandscape File

The BidLandscape class takes as input a request rate (requests/hour), mean and variance of winning prices and ctr and calculates and holds the corresponding log-normal winning price distribution and interfaces such as the winning function.  
The class further provides a mix method which can be used to mix multiple landscapes in an appropriate way (used e.g. for using both training data and previous historical data to determine the expected bid landscape)

The file also includes the BidLandscapeServer, which retrieves the BidLandscape initialization parameters of the training set dynamically, either from Spark or from memory if they have been loaded before. The code should be self-explanatory.

# WP-Prediction Pipeline

The main script for WP-Prediction is the WPPrediction.scala script, which follows the procedure

1. Load the raw data
2. Preprocessing
3. FeaturePreparation
4. Model Training
5. Model Evaluation
6. Logging
7. (optional, commented out) Write the simulation data to disk using the current WP-prediction model.

In steps 2,3 and 4 the additional objects Preprocessing, FeaturePreparation and Training are called.

In the following, we’ll briefly explain what happens in each of these steps and where parameters/options can be set.

## 1 Loading the Data

The script can either load the full training data set (/data\_processed\_0908\_with\_CTR/Full/) or one of the samples in /data\_processed\_0908\_with\_CTR/Samples/. This behavior can be set in line 41 of the script.

## 2 Preprocessing

As a next step, the script calls the Preprocessing object on the raw input data. At a previous point in development, additional conversions and data cleanup was performed here, such as mapping indices to values in several columns. Many of these steps have become obsolete in the final version due to updates applied to the dataset itself (provided by Yannis). The only remaining active step in the final implementation is mapping the raw data frame into a corresponding schema, such that spark knows the data types of each column without having to infer them automatically (and thus traverse the entire dataset once).

### 3 Feature Preparation

This is the main source of variability. The script will next call the FeaturePreparation object, which determines how features will be seen by the machine learning algorithm in the next step. In the FeaturePreparation.scala script, the user can determine for each column to be treated as one of the following types:

|  |  |
| --- | --- |
| **Type** | **Comment** |
| Numerical Column | These columns should contain interval scaled values and will enter the ML model directly as features. |
| Categorical Column Safe | These columns should contain categorical attributes with a known (usually small) set of possible values. “Safe” here refers to the requirement, that all possible values should be contained in the training set. These columns will be one-hot encoded after using Spark’s standard StringIndexer which cannot handle unseen values in the training set. |
| Categorical Column Unsafe | Similar to the previous, except able to handle unseen values at somewhat greater compuatational cost. Will be one-hot encoded after using custom MyStringIndexer which extends Spark’s standard StringIndexer. |
| Categorical Column Hashed | Categorical columns in this category will not be One-Hot-Encoded, but instead be Hashed using the FeatureHashing trick as implemented in FeatureHashing.scala This can be seen as an alternative to “unsafe” OHE, which sacrifices some accuracy in order to gain massive amounts of speed and a lower dimensionality, which can manually be set in the FeaturePreparation script. |
| UT Column | The UT-columns are all Boolean and will simply be used as Boolean features. |

Once the column types are defined, the dataset will then go through the following pipeline:

|  |  |
| --- | --- |
| **Stage** | **Comment** |
| 1 numVecAssembler | Assembles all numerical columns into a dense spark vector FT\_num |
| 2 ftHasher | Hashes the Hashing-Columns into a sparse spark vector FT\_hashed with dimensionality numHashed |
| 3 StringIndexersSafe | Applies Spark’s StringIndexer to each safe numerical column, producing columns ID\_*colname* of indices |
| 4 StringIndexersUnsafe | Applies MyStringIndexer to each unsafe num. column, producing columns ID\_*colname* of indices |
| 5 oheEncoders | OH-Encodes all ID\_*xxx* columns, producing columns FT\_*xxx*\_*index* |
| 6 utVecAssembler | Assembles UT columns into a sparse spark vector FT\_ut |
| 7 ftAssembler | Assembles all FT\_*xxx* vectors into a combined vector “features” |

## 4 Training

With the features created, the Training part of the script is then called. First, a train-valuation split is set up in the main script. Then the Training.scala script is called on the training part of the data.

In this file, we set which model should be trained as well as the corresponding ML hyperparameters. For GBDT models, this is the number of Iterations, i.e. the number of sequential trees to grow (we do not change the loss function (squared), model type (regression) as these are uniquely appropriate for our problem, similarily we do not tune the learning rate, as suggested in the Spark documentation.) For LR models, we experiment with the max. number of iterations (usually we reach convergence when set to 100 or more), as well as the regularization parameters (ElasticNetParam, RegParam).

## 5 Evaluation

We evaluate the model trained in 4 on the valuation set to determine its RMSE as well as possible overfitting.

## 6 Logging

A log file is written to disk, containing all FeaturePreparation and Training parameters as well as the results.

## 7 (Optional) Writing Simulation Data

If enabled, use the model trained in 4 to make WP predictions on the entire iPinYouSeason2 test set and write the results to disk in the form required by the simulation framework.