Database Management Systems

Lecture 12

Distributed Databases (II)

* Azure Machine Learning *

Distributed Databases

Distributed Catalog Management

- schema, authorization information, statistics
- keeping track of data distribution across sites
- identify each replica of each fragment for a relation that is fragmented and replicated
- local autonomy should not be compromised:
  - global relation name:
    - <local-name, birth-site>
  - global replica name:
    - <local-name, birth-site, replica-id>

Distributed Catalog Management

- centralized system catalog
- global system catalog maintained at each site
- local catalog maintained at each site

- centralized system catalog
    - stored at a single site
    - contains data about all the relations, fragments, replicas
    - vulnerable to single-site failures
    - can overload the server

Distributed Catalog Management

- global system catalog maintained at each site
  - every copy of the catalog describes all the data
  - not vulnerable to single-site failures (the data can be obtained from a different site)
  - local autonomy is compromised:
    - changes to a local catalog must be propagated to all the other sites

Distributed Catalog Management

- local catalog maintained at each site
  - each site keeps a catalog that describes local data, i.e., copies of data stored at the site
  - the catalog at the birth site for a relation keeps track of all the fragments / replicas of the relation
  - create a new replica / move a replica to another site:
    - must update the catalog at the birth site
  - not vulnerable to single-site failures
  - doesn't compromise local autonomy

Distributed Transaction Management

- a transaction submitted at a site S could ask for data stored at several other sites
- *subtransaction*
    - the activity of a transaction at a given site
- context
    - Strict 2PL with deadlock detection
- problems
    - distributed concurrency control
    - distributed recovery

Distributed Transaction Management

- distributed concurrency control
    - objects stored across several sites - lock management
    - deadlock detection
- distributed recovery
    - transaction atomicity
        - all the effects of a committed transaction (across all the sites it executes at) are permanent
        - none of the actions of an aborted transaction are allowed to persist

Distributed Transaction Management

- distributed concurrency control
    - lock management
        - techniques – synchronous / asynchronous replication
            - which objects will be locked
        - concurrency control protocols
            - when are locks acquired / released
        - approaches
            - *centralized*
            - *primary copy*
            - *fully distributed*

Distributed Transaction Management

- distributed concurrency control
  - lock management
    - centralized
      - one site does all the locking for all the objects
      - vulnerable to single-site failures
    - primary copy
      - object O, PC - primary copy of O stored at site S with lock manager L
      - all requests to lock / unlock a copy of O are handled by L
      - not vulnerable to single-site failures
      - read copy C of O stored at site S2:
        => communicate with both S and S2

Distributed Transaction Management

- distributed concurrency control
  - lock management
    - fully distributed
      - object O, C - copy of O stored at site S with Lock Manager L
        - requests to lock / unlock C are handled by L (the site where the copy is stored)
        - read a copy of O - don't need to access 2 sites

Distributed Transaction Management

- distributed concurrency control
  - detect and resolve deadlocks
  - each site maintains a local waits-for graph
  - a cycle in such a graph indicates a deadlock
  - but a deadlock can exist even if none of the local graphs contains a cycle

->

Distributed Transaction Management

- distributed concurrency control – distributed deadlock
    - e.g., using *read-any write-all*

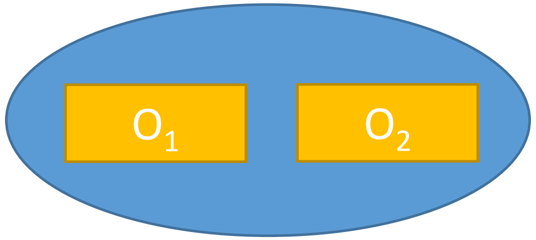
- T_1 wants to read O_1 and write O_2
- T_2 wants to read O_2 and write O_1

- T_1 acquires an S lock on O_1 and an X lock on O_2 at site S1
- T_2 obtains an S lock on O_2 and an X lock on O_1 at site S2

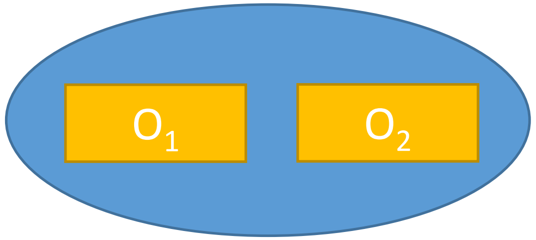
- T_1 requests an X lock on O_2 at site S2
- T_2 requests an X lock on O_1 at site S1

Distributed Transaction Management

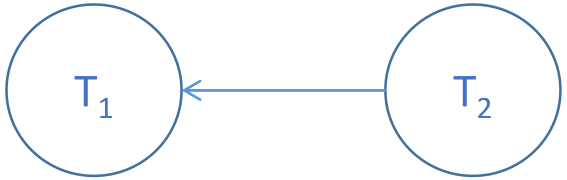
- distributed concurrency control – distributed deadlock
  - e.g., using *read-any write-all*



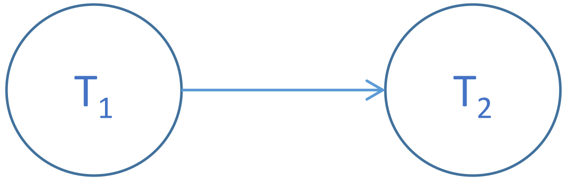
site S1



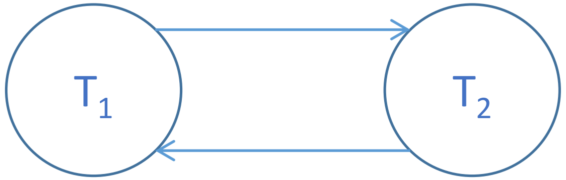
site S2



at site S1



at site S2



global waits-for graph

Distributed Transaction Management

- distributed concurrency control – distributed deadlock
  - distributed deadlock detection algorithms
    - *centralized*
    - *hierarchical*
    - based on a *timeout* mechanism

Distributed Transaction Management

- distributed concurrency control – distributed deadlock
    - distributed deadlock detection algorithms
        - centralized
            - all the local waits-for graphs are periodically sent to a single site S
            - S - responsible for global deadlock detection
            - the global waits-for graph is generated at site S
                - nodes
                    - the union of nodes in the local graphs
                - edges
                    - there is an edge between 2 nodes if such an edge exists in one of the local graphs

Distributed Transaction Management

- distributed concurrency control – distributed deadlock
  - distributed deadlock detection algorithms
    - hierarchical
      - sites are organized into a hierarchy, e.g., grouped by city, country, etc
      - each site periodically sends its local waits-for graph to its parent site
      - assumption: more deadlocks are likely across related sites
      - all the deadlocks are detected in the end

Distributed Transaction Management

- distributed concurrency control – distributed deadlock
  - distributed deadlock detection algorithms
    - hierarchical
      - example:

        RO (CJ (Cluj-Napoca, Dej, Turda), BN (Bistrita, Beclean))

        Cluj-Napoca : T1 -> T2
        Dej: T2 -> T3
        Turda: T3 -> T4 <- T7
        Bistrita: T5 -> T6
        Beclean: T4 -> T7 -> T6 -> T5

        CJ: T1 -> T2 -> T3 -> T4 <- T7
        BN: T5 <-> T6 <- T7 <- T4 (*)

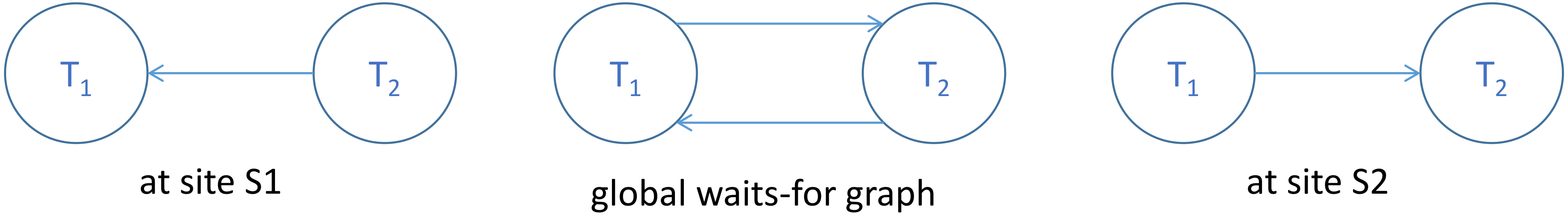
        RO: T1 -> T2 -> T3 -> T4 <-> T7 -> *T6* <-> *T5*
        Obs RO: T5 or T6 aborted at (*)

Distributed Transaction Management

- distributed concurrency control – distributed deadlock
  - distributed deadlock detection algorithms
    - based on a timeout mechanism
      - a transaction is aborted if it lasts longer than a specified interval
      - can lead to unnecessary restarts
      - however, the deadlock detection overhead is low
      - could be the only available option in a heterogeneous system (if the participating sites cannot cooperate, i.e., they cannot share their waits-for graphs)

Distributed Transaction Management

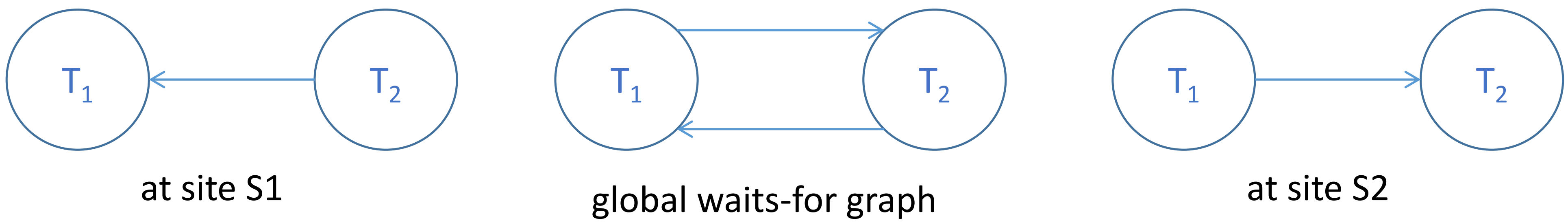
- distributed concurrency control – distributed deadlock
- phantom deadlocks
    - "deadlocks" that don't exist, but are detected due to delays in propagating local information
    - lead to unnecessary aborts
    - example:



- generate local waits-for graphs at sites S1 and S2
- send local waits-for graphs to the site responsible for global deadlock detection

Distributed Transaction Management

- distributed concurrency control – distributed deadlock
- phantom deadlocks
    - example:



- T2 aborts (not because of the deadlock)

=> local waits-for graphs are changed, there is no cycle in the "real" global waits-for graph

- but the built global waits-for graph does have a cycle
- T1 could be chosen as a victim

Distributed Transaction Management

- distributed recovery
  - more complex than in a centralized DBMS
  - new types of failure
    - network failure
    - site failure
  - commit protocol
    - either all the subtransactions of a transaction commit, or none of them does
  - normal execution
    - ensure all the necessary information is provided to recover from failures
  - a log is maintained at each site
    - data logged in a centralized DBMS
    - actions carried out as part of the commit protocol

Distributed Transaction Management

- distributed recovery
    - transaction T
        - coordinator
            - the Transaction Manager at the site where T originated
        - subordinates
            - the Transaction Managers at the sites where T's subtransactions execute

Distributed Transaction Management

- distributed recovery
  - Two-Phase Commit (2PC) protocol
    - exchanged messages, records written in the log
    - 2 rounds of messages, both initiated by the coordinator
      - *voting phase*
      - *termination phase*
    - any Transaction Manager can abort a transaction
    - however, for a transaction to commit, all Transaction Managers must decide to commit

Distributed Transaction Management

- distributed recovery
  - Two-Phase Commit protocol
    - the user decides to commit transaction T

    => the commit command is sent to T's coordinator, initiating 2PC

    1. the coordinator sends a *prepare* message to each subordinate
    2. upon receiving a *prepare* message, a subordinate decides whether to commit / abort its subtransaction
       - the subordinate:
         - force-writes an *abort* or a *prepare** log record
         - then sends a *no* or *yes* message to the coordinator

* *prepare* log records are specific to the commit protocol, they are not used in centralized DBMSs

Distributed Transaction Management

- distributed recovery
  - Two-Phase Commit protocol

    3.
    - if the coordinator receives *yes* messages from all subordinates, it:
      - force-writes a *commit* log record
      - then sends a *commit* message to all subordinates
    - otherwise (i.e., if it receives at least one *no* message / it doesn't receive any message from a subordinate for a specified timeout interval):
      - it force-writes an *abort* log record
      - it then sends an *abort* message to each subordinate

Distributed Transaction Management

- distributed recovery
  - Two-Phase Commit protocol

    4.
    - upon receiving an *abort* message, a subordinate:
      - force-writes an *abort* log record
      - sends an *ack* message to the coordinator
      - aborts the subtransaction
    - upon receiving a *commit* message, a subordinate:
      - force-writes a *commit* log record
      - sends an *ack* message to the coordinator
      - commits the subtransaction

Distributed Transaction Management

- distributed recovery
    - Two-Phase Commit protocol

      5. after it receives *ack* messages from all subordinates, the coordinator writes an *end* log record for the transaction

      * obs. sending a message
        - the sender has made a decision
        - the message is sent only after the corresponding log record has been forced to stable storage (to ensure the corresponding decision can survive a crash)

Distributed Transaction Management

- distributed recovery
    - Two-Phase Commit protocol
        - log records for the commit protocol
            - record type
            - transaction id
            - coordinator's identity
        - the commit / abort log record of the coordinator also contain the identities of the subordinates

        - committed transaction
            - T is a committed transaction when the commit log record of T's coordinator is forced to stable storage

Distributed Transaction Management

- distributed recovery
  - restart after a failure – site S comes back up after a crash
    - if there is a *commit* or an *abort* log record for transaction T:
      - must redo / undo T
      - if S is T's coordinator:
        - periodically send *commit* / *abort* messages to subordinates until *ack* messages are received
        - write an *end* log record after receiving all *ack* messages
    - no *commit* / *abort* log records, but there is a *prepare* log record for T => S is one of T's subordinates
        - contact T's coordinator repeatedly until T's status is obtained
        - write a *commit* / an *abort* log record
        - redo / undo T

Distributed Transaction Management

- distributed recovery
  - restart after a failure – site S
    - if there are no *commit / abort / prepare* log records for T:
      - abort, undo T
      - if S is T's coordinator, T's subordinates may subsequently contact S

Distributed Transaction Management

- distributed recovery
    - restart after a failure – site S

      *obs. blocking
        - T's coordinator site fails
            - T's subordinates who have voted *yes* cannot decide whether to commit or abort T until the coordinator recovers, i.e., T is *blocked*
            - include the ids of subordinates in the *prepare* messages
                - the subordinates can communicate with each other
                - even if all subordinates voted *yes*, the coordinator's decision cannot be determined until it recovers

Distributed Transaction Management

- distributed recovery
  - link and remote site failures
    - current site S, remote site R, transaction T
    - if R doesn't respond during the commit protocol for T (site / link failure):
      - if S is T's coordinator:
        - S should abort T
      - if S is one of T's subordinates, and has not voted yet:
        - S should abort T
      - if S is one of T's subordinates and has voted *yes:*
        - S is blocked until T's coordinator responds

Distributed Transaction Management

- distributed recovery

  * 2PC – obs

  - *ack* messages
    - used to determine when can a coordinator C "forget" about a transaction T
    - C must keep T in the transaction table until it receives all *ack* messages
  - coordinator C fails after sending *prepare* messages, but before writing a *commit* / an *abort* log record
    - when C comes back up, it aborts T
    - i.e., absence of information => T is presumed to have aborted
  - if a subtransaction doesn't change any data, its commit / abort status is irrelevant

Distributed Transaction Management

- distributed recovery
  - 2PC with Presumed Abort
    - coordinator C, transaction T, subordinate S, subtransaction t
      - C aborts T
        - T is undone
        - C immediately removes T from the Transaction Table, i.e., it doesn’t wait for *ack* messages
      - subordinates' names need not be recorded in C's abort log record
      - S doesn't need to send an *ack* message when it receives an *abort* message

Distributed Transaction Management

- distributed recovery
  - 2PC with Presumed Abort
    - coordinator C, transaction T, subordinate S, subtransaction t
      - t doesn't change any data
        - the subordinate responds to a *prepare* message with a *reader* message, instead of a *yes* / *no*
      - after receiving a *reader* message, C doesn't send any other messages to the subordinate
      - if all subtransactions are readers, the 2nd phase of the protocol is not needed

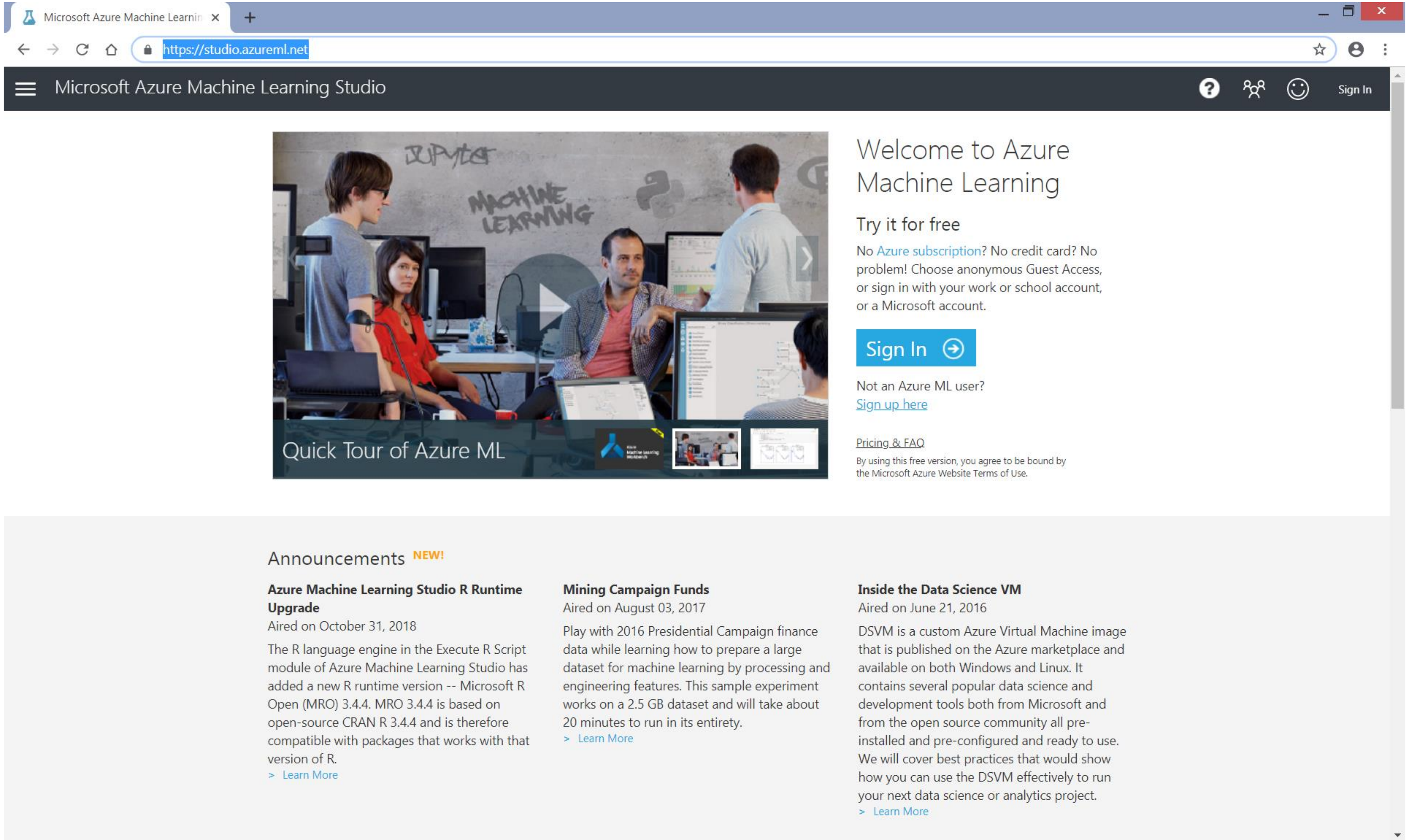
References

- [Ra00] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (2nd Edition), McGraw-Hill, 2000
- [Da03] DATE, C.J., An Introduction to Database Systems (8th Edition), Addison-Wesley, 2003
- [Ga08] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book, Prentice Hall Press, 2008
- [Ra07] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, McGraw-Hill, 2007, http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html
- [Si10] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts, McGraw-Hill, 2010, http://codex.cs.yale.edu/avi/db-book/
- [Ul11] ULLMAN, J., WIDOM, J., A First Course in Database Systems, http://infolab.stanford.edu/~ullman/fcdb.html

Azure Machine Learning

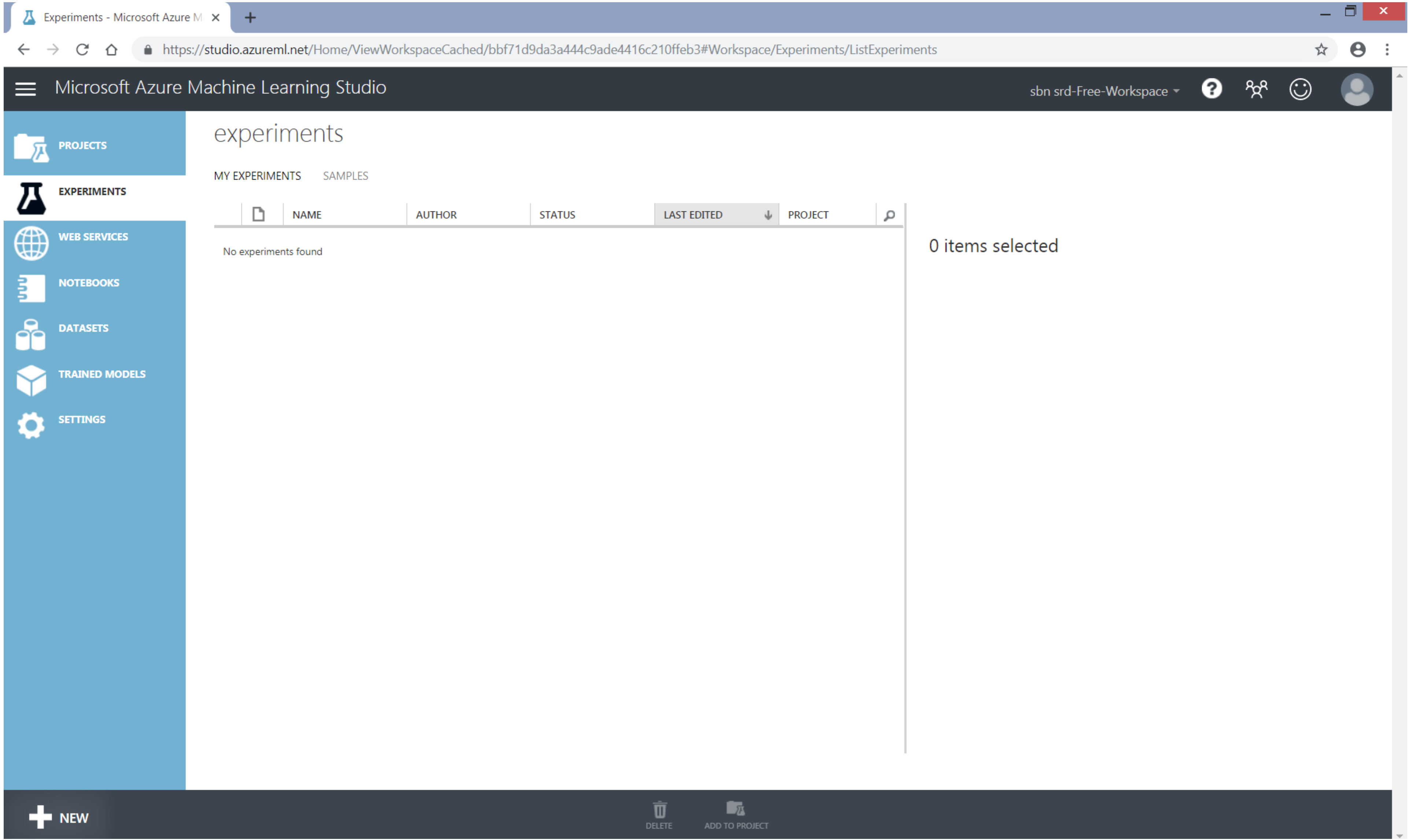
Data science Experiment – Car Price Prediction

- https://studio.azureml.net/ -> Sign in



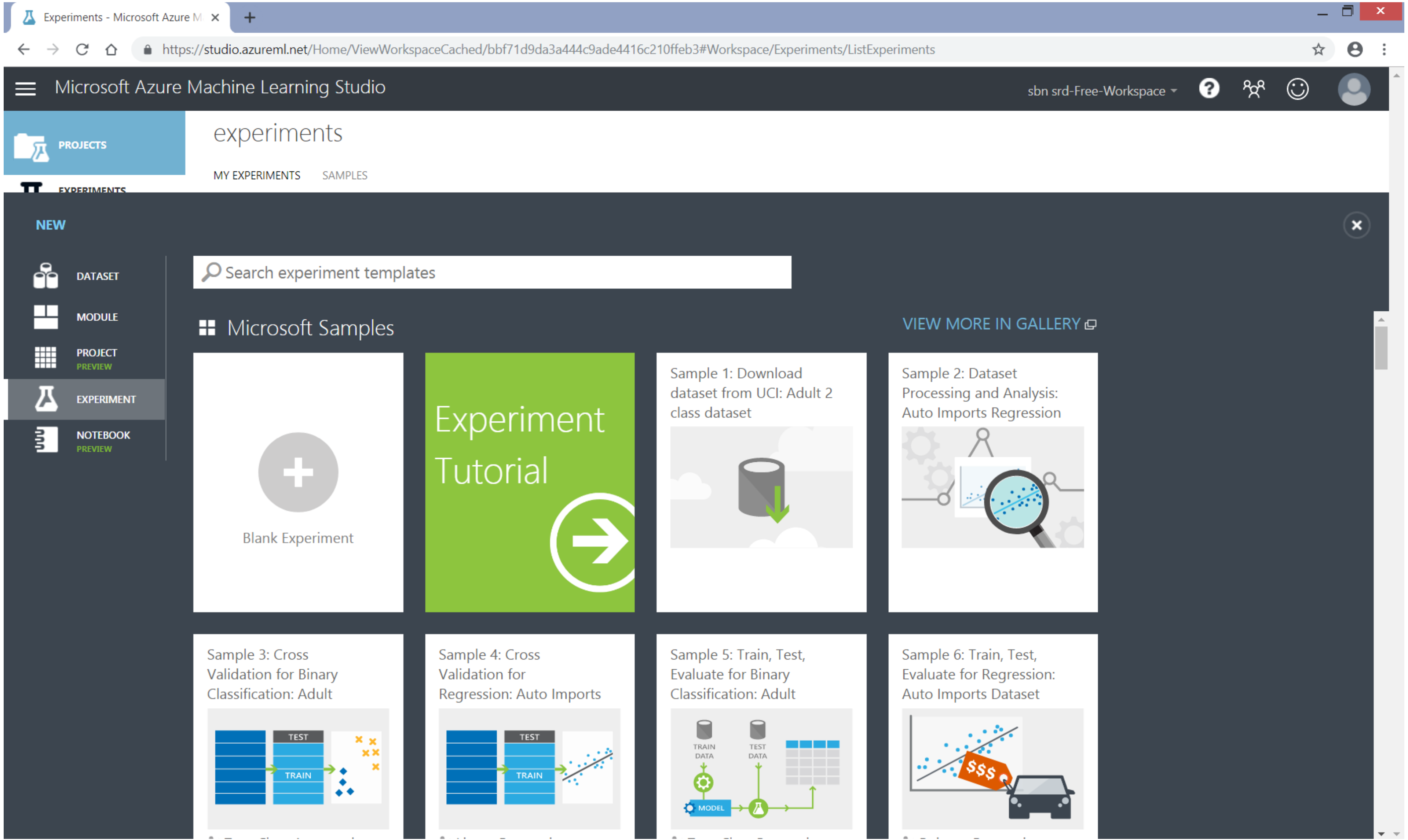
Car Price Prediction

* create an experiment: + New



Car Price Prediction

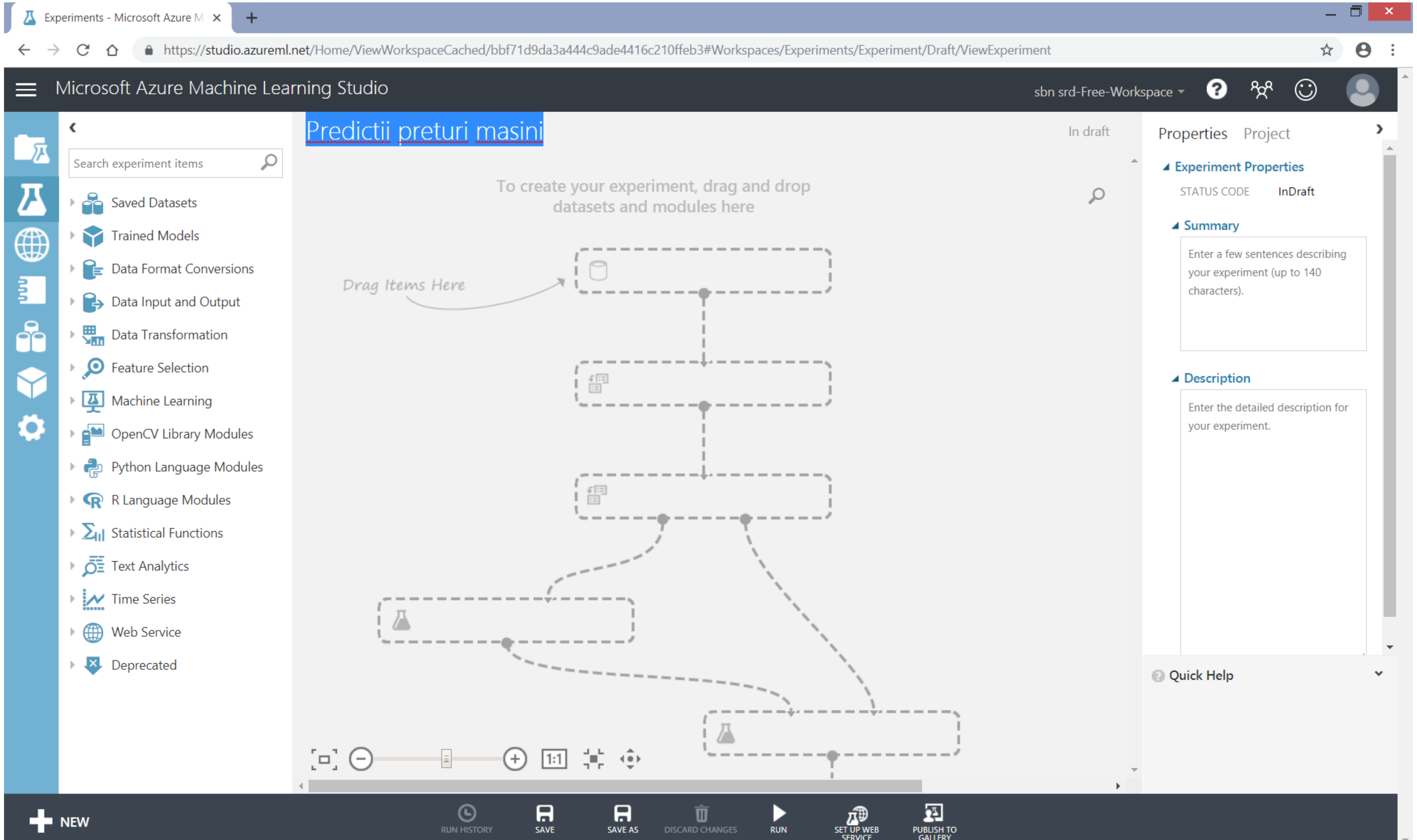
* create an experiment: Blank Experiment



Car Price Prediction

* create an experiment

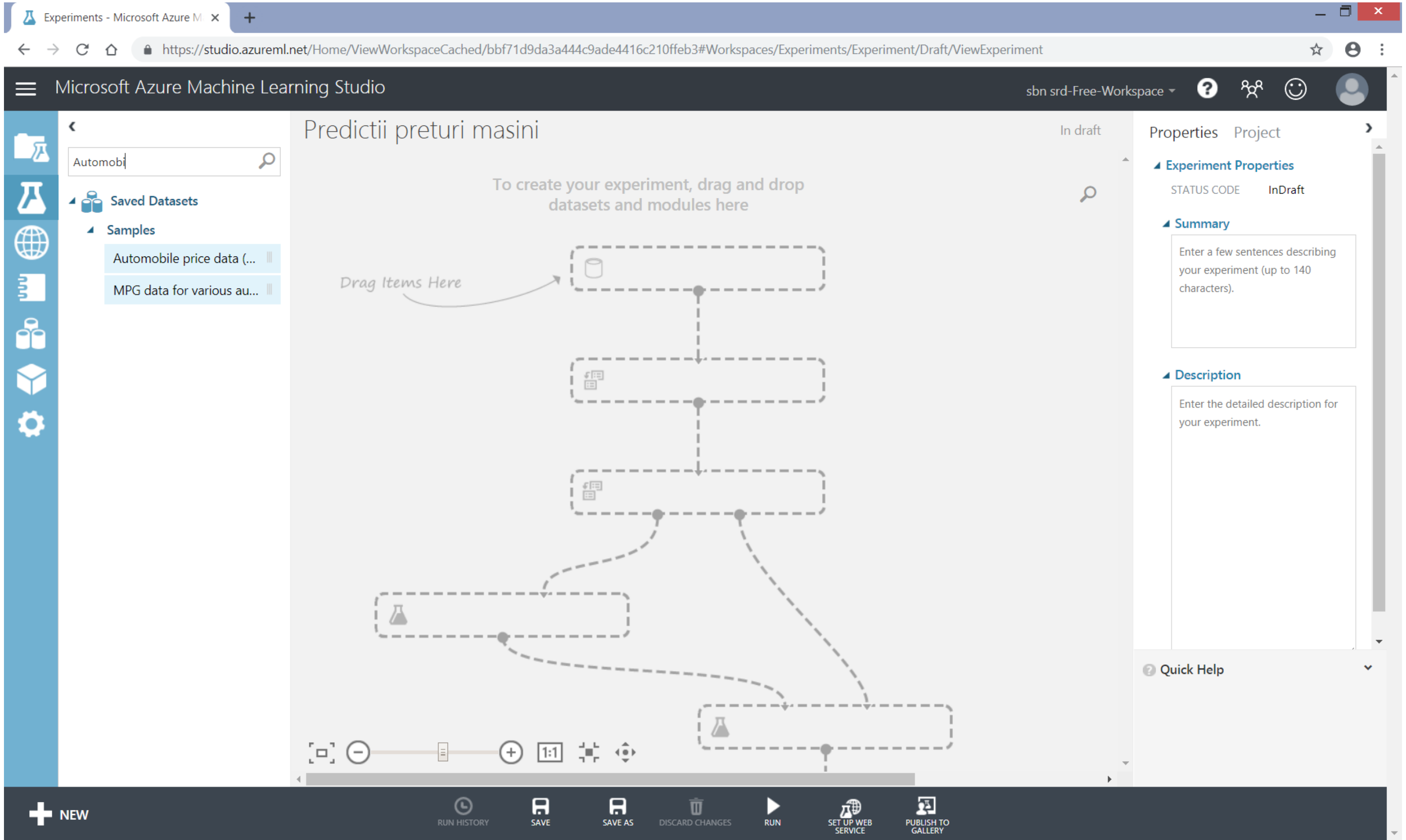
- experiment name



Car Price Prediction

* selecting the data source

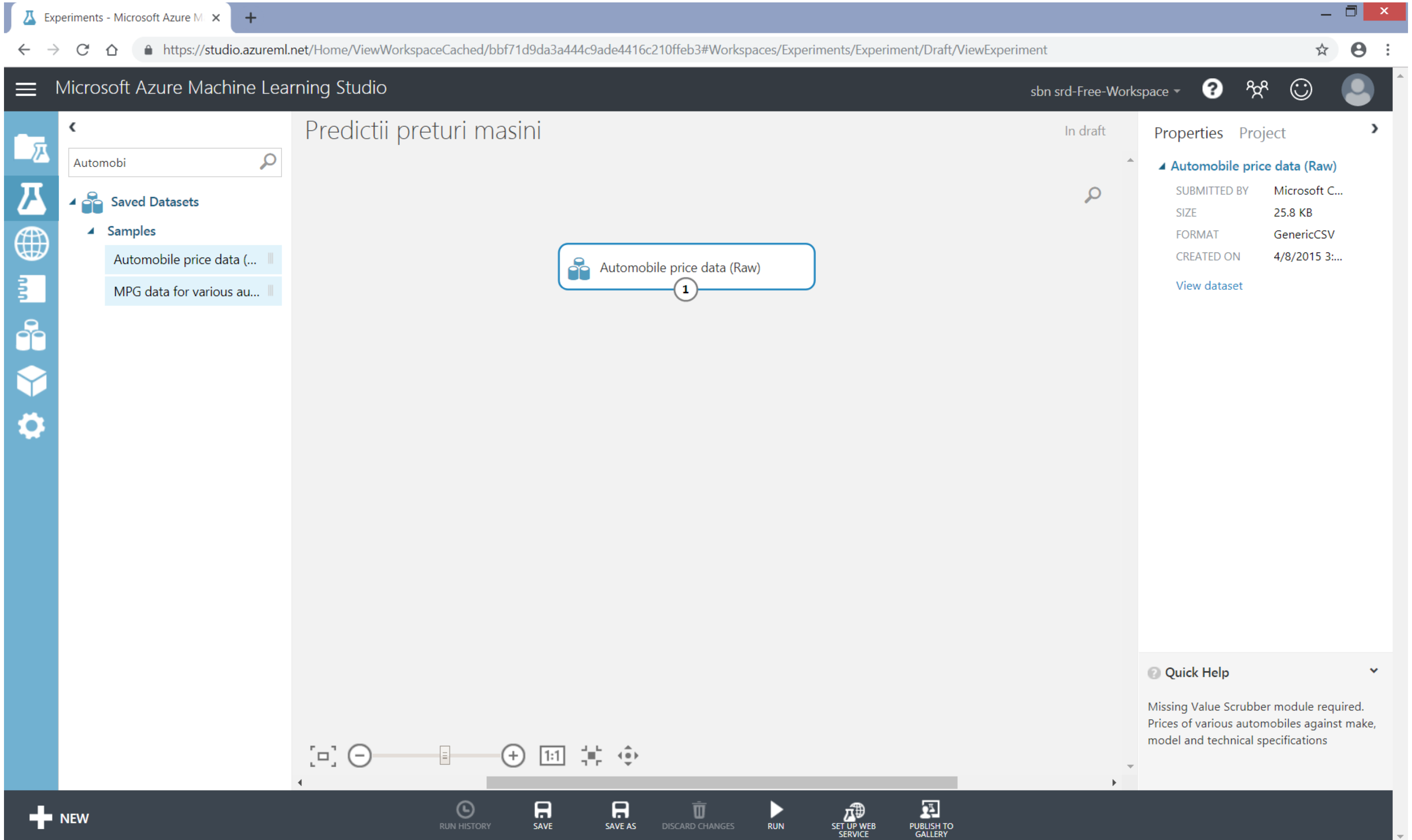
- dataset *Automobile price data (raw)*



Car Price Prediction

* selecting the data source

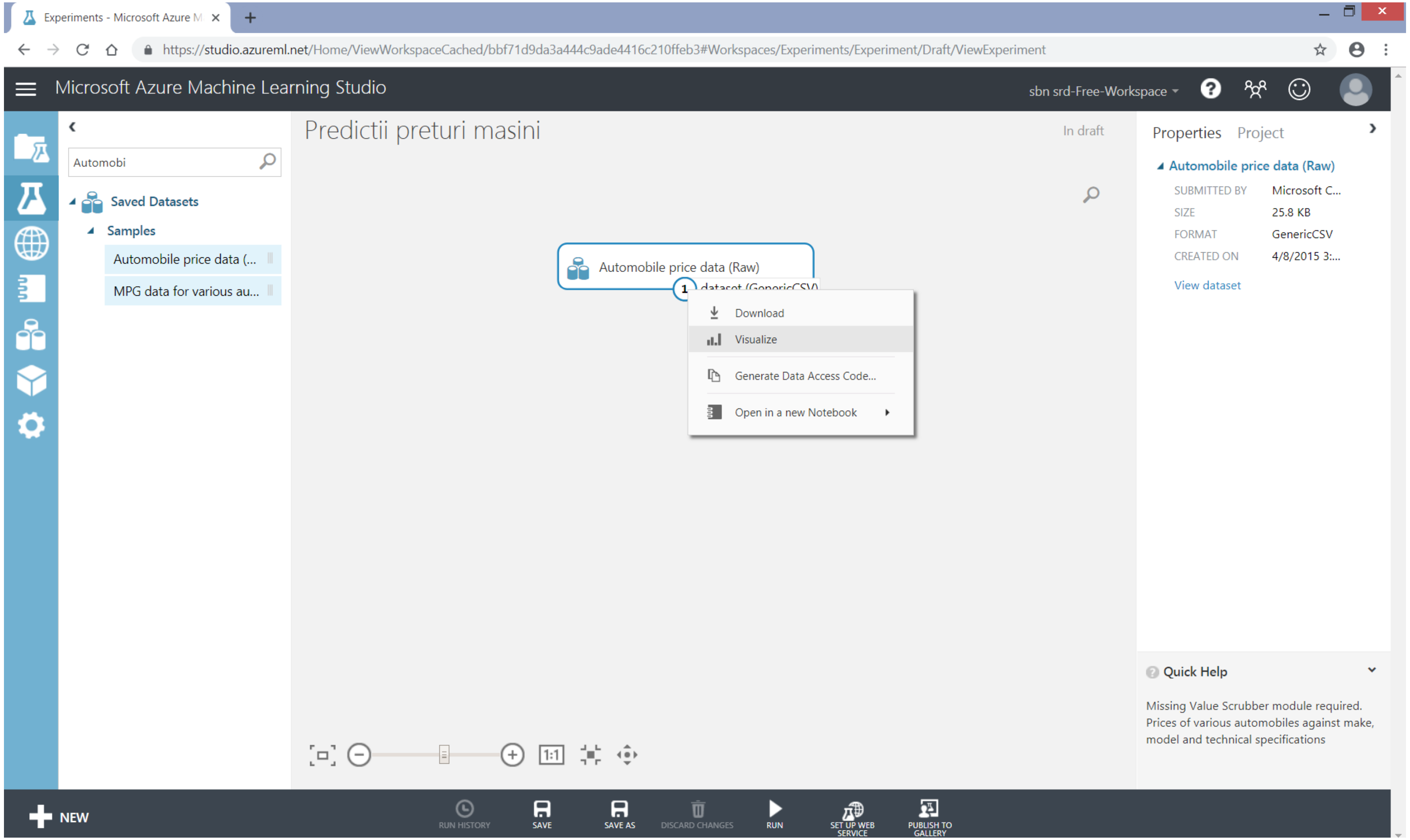
- drag & drop dataset *Automobile price data (raw)* onto the canvas



Car Price Prediction

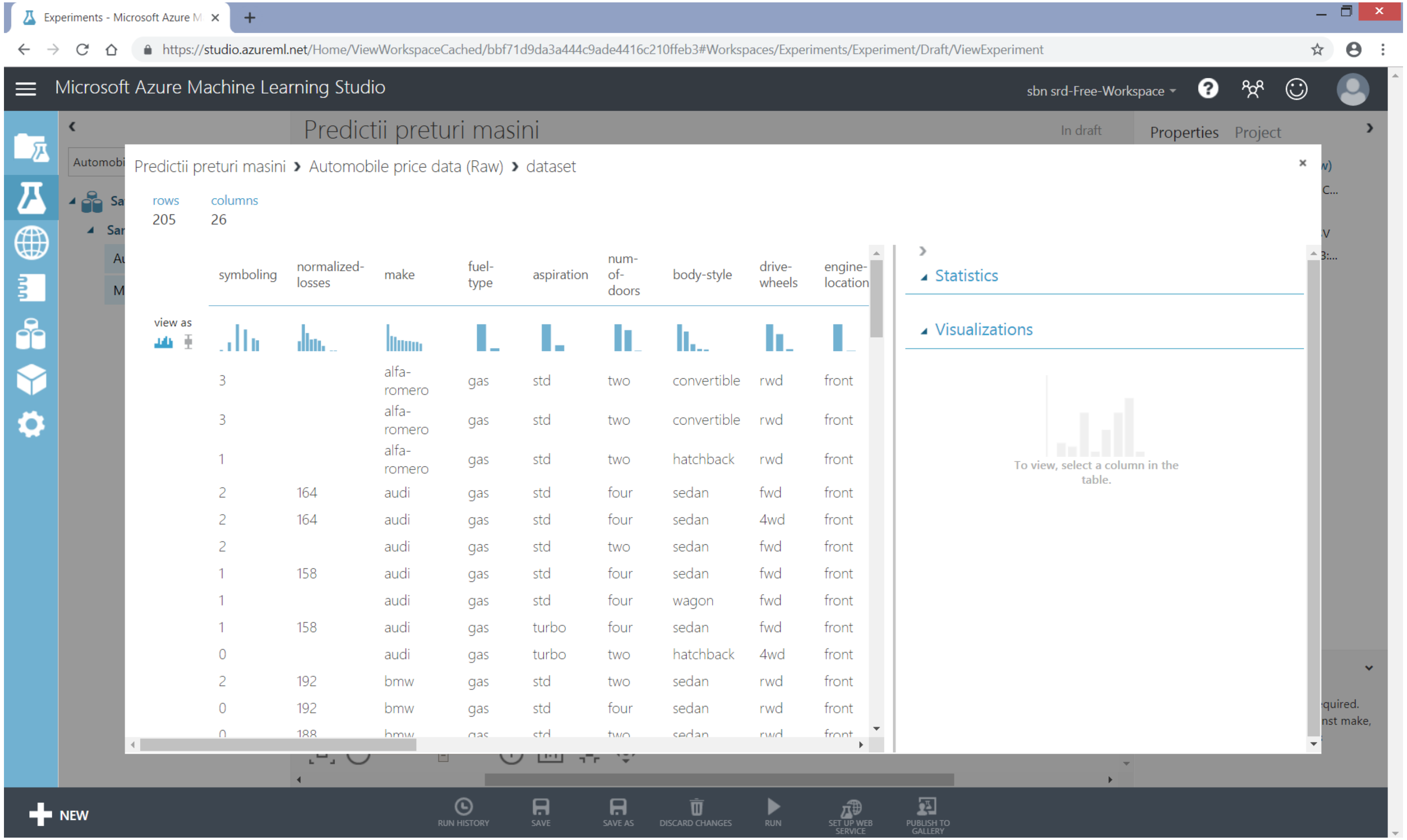
* displaying the data

- dataset output port -> *Visualize*



Car Price Prediction

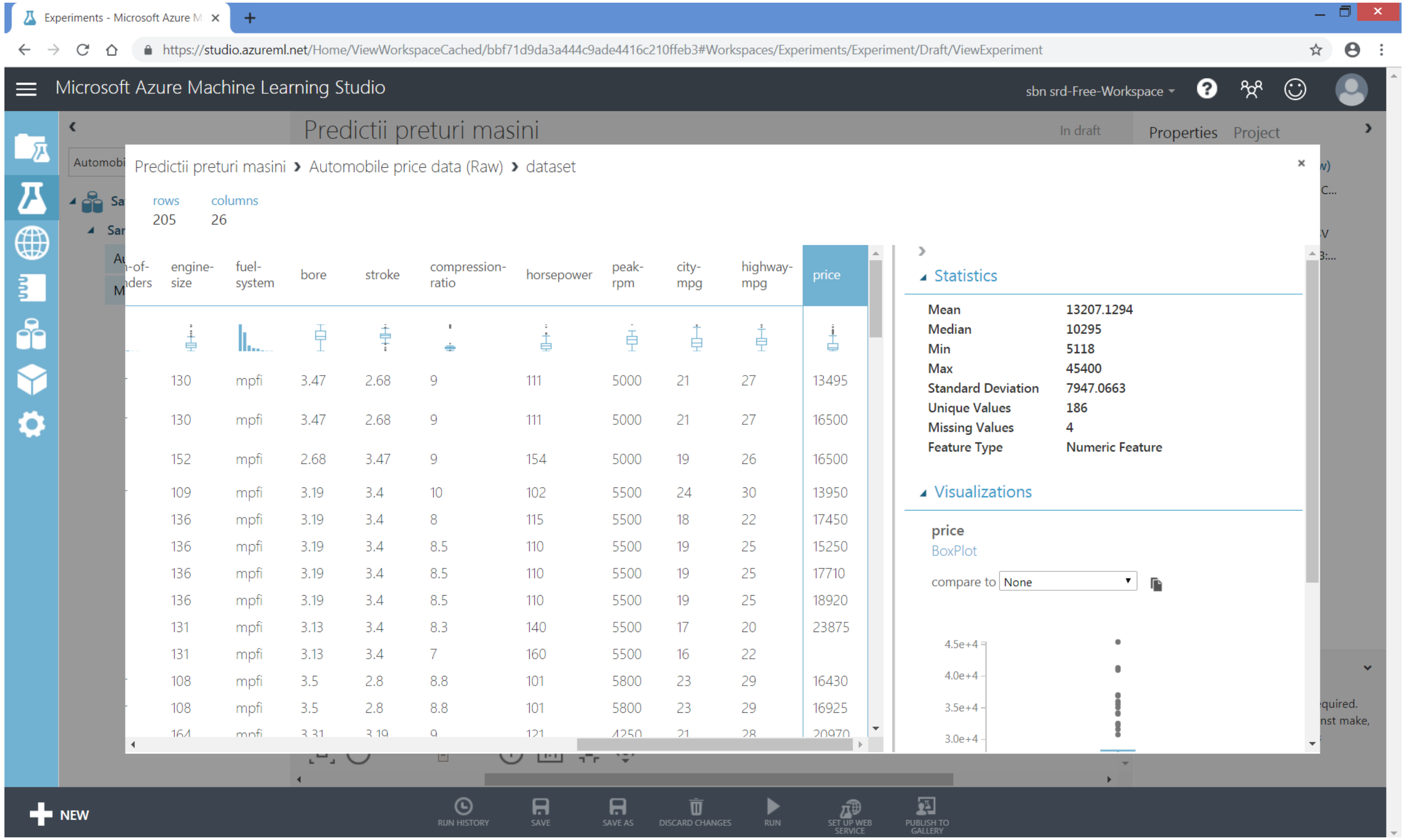
* displaying the data
    - row – data about a car



Car Price Prediction

* displaying the data

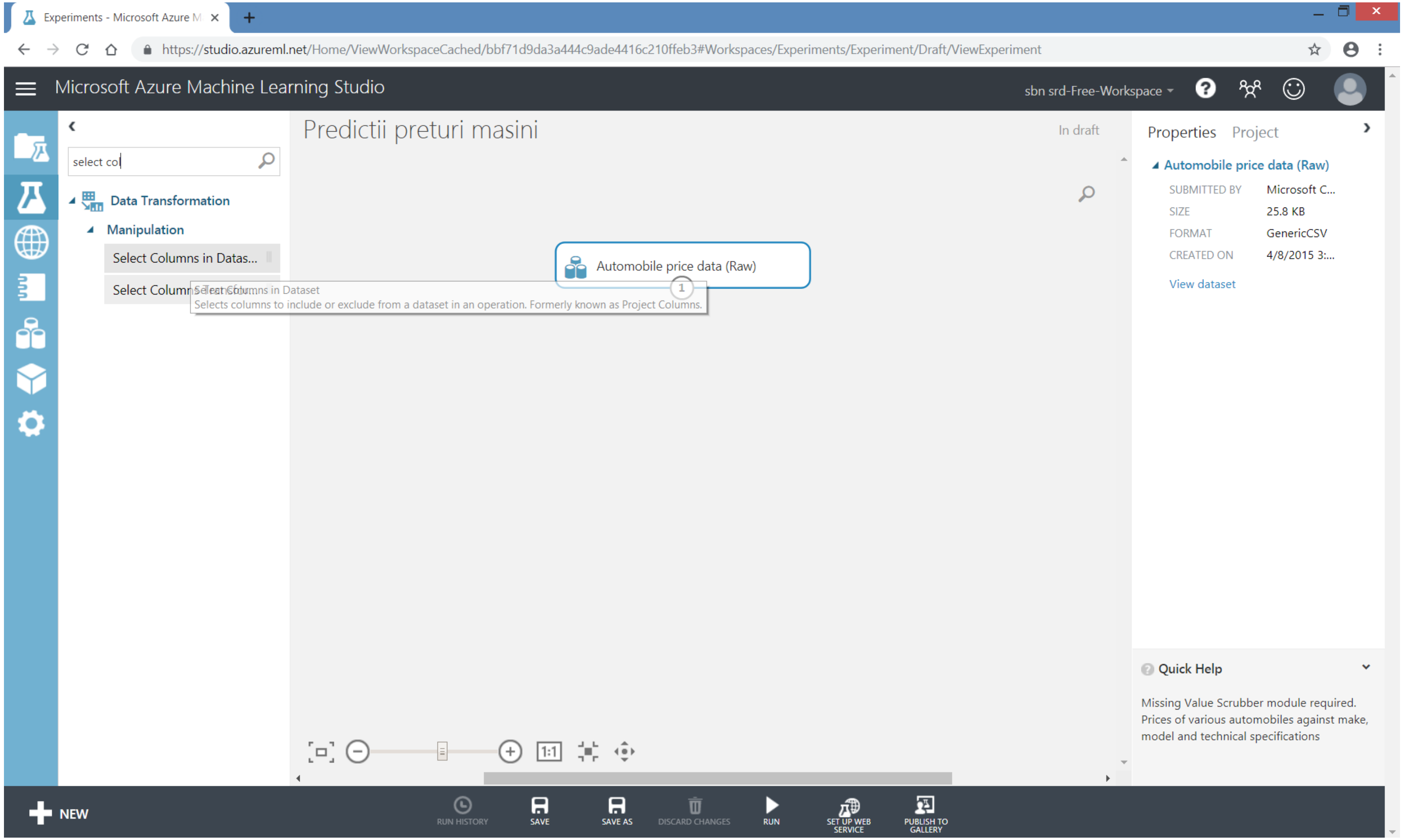
- columns – variables
- *target* column - *price*



Car Price Prediction

* preparing the data

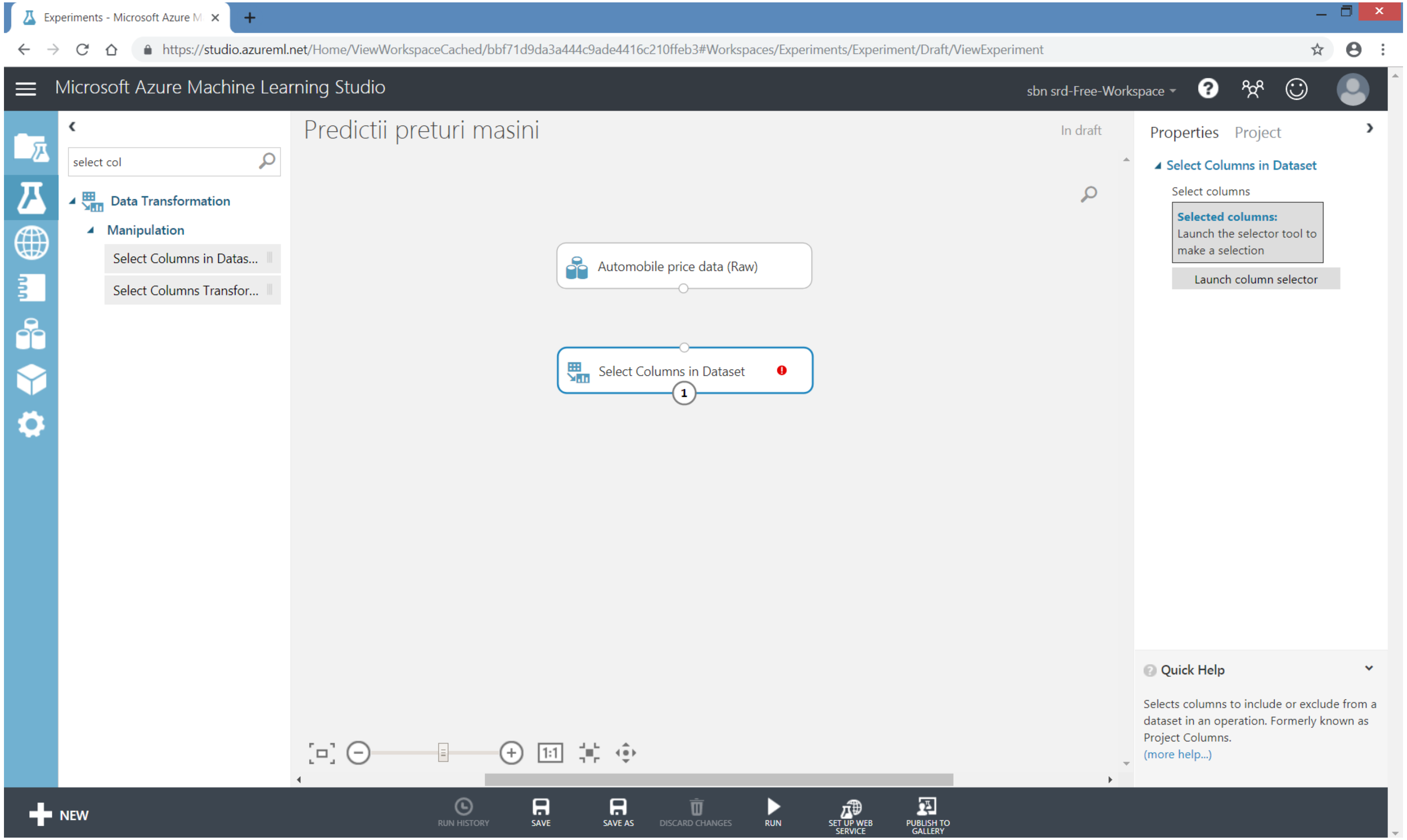
- eliminate column with missing values – *normalized-losses*



Car Price Prediction

* preparing the data

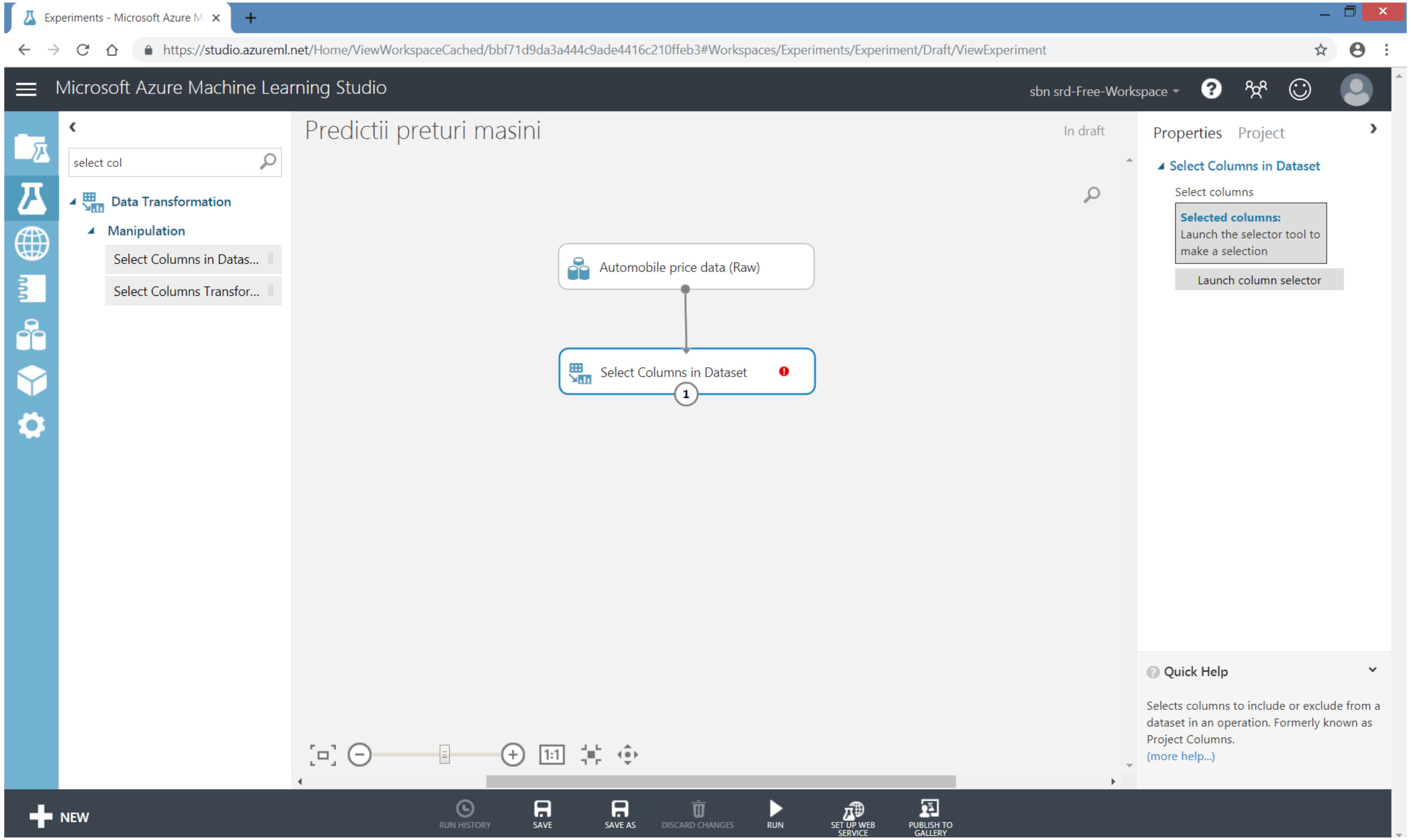
- eliminate column with missing values - *Select Columns in Dataset* module



Car Price Prediction

* preparing the data

- eliminate column with missing values

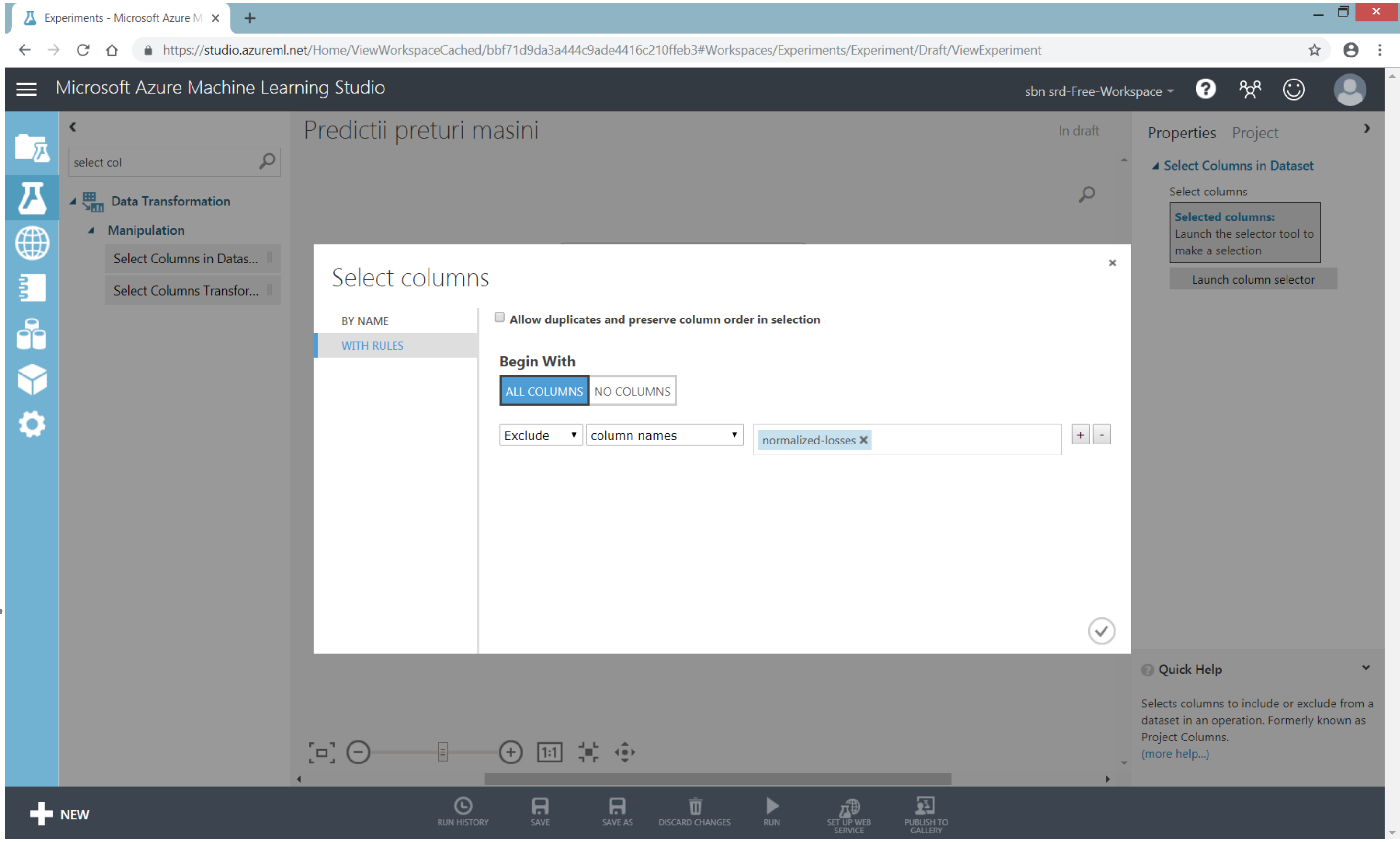


Car Price Prediction

* preparing the data

- eliminate column with missing values

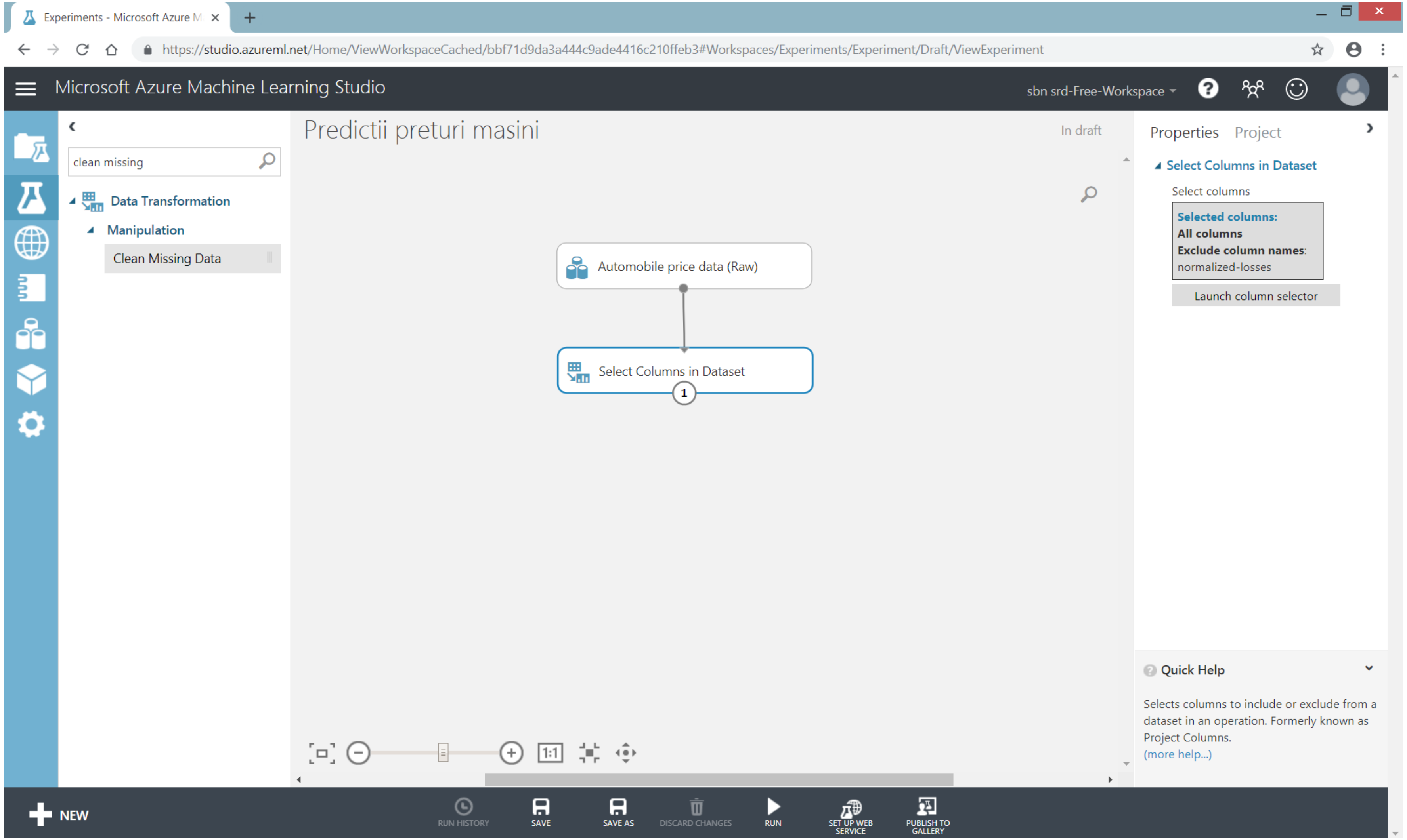
- *Select Columns in Dataset*
  - *Launch column selector*
    - *With Rules*
      - *Begin With*
        - *All Columns*
      - *Exclude*
        - *normalized-losses*



Car Price Prediction

* preparing the data

- eliminate rows with missing values – *Clean Missing Data* module

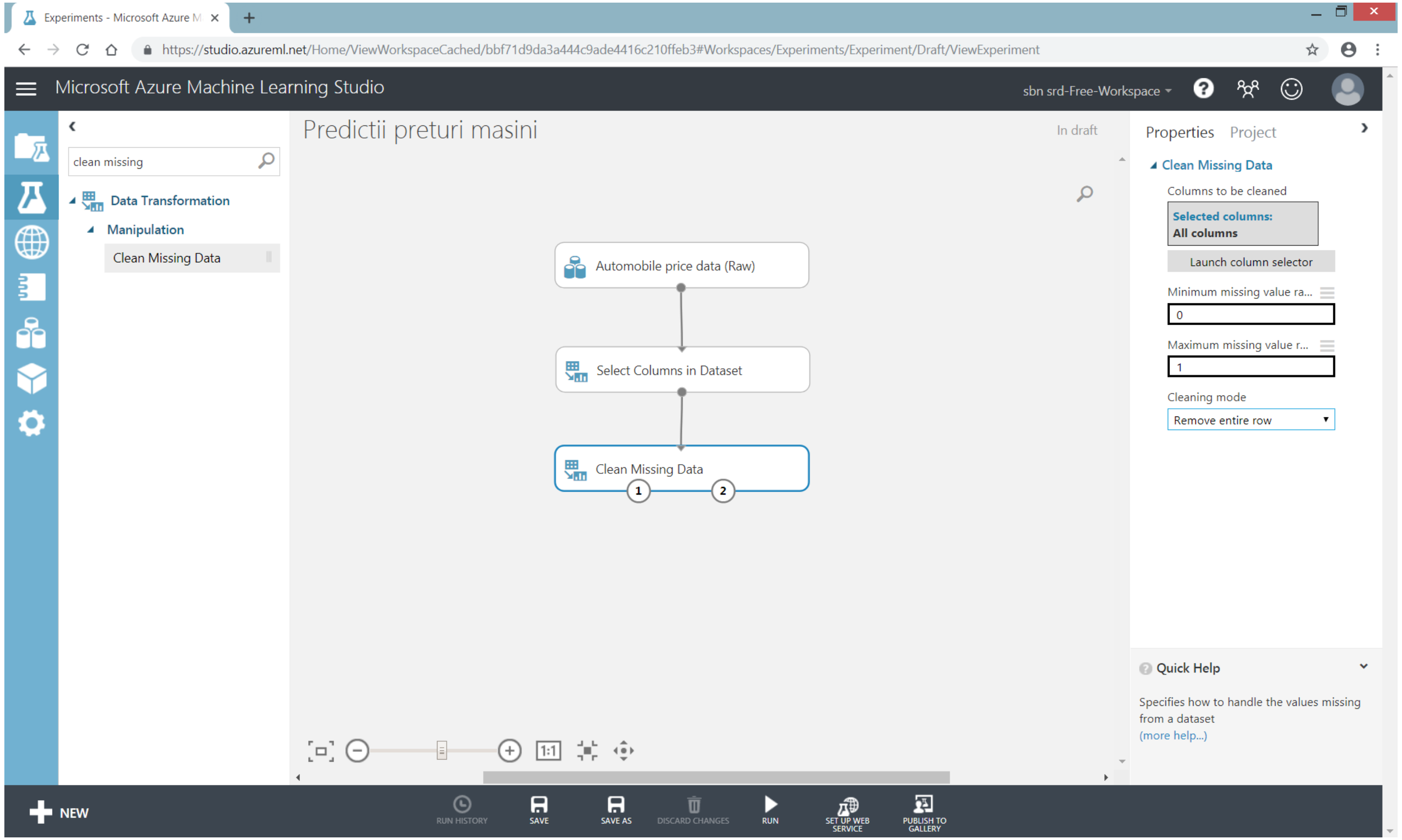


Car Price Prediction

* preparing the data

- eliminate rows with missing values

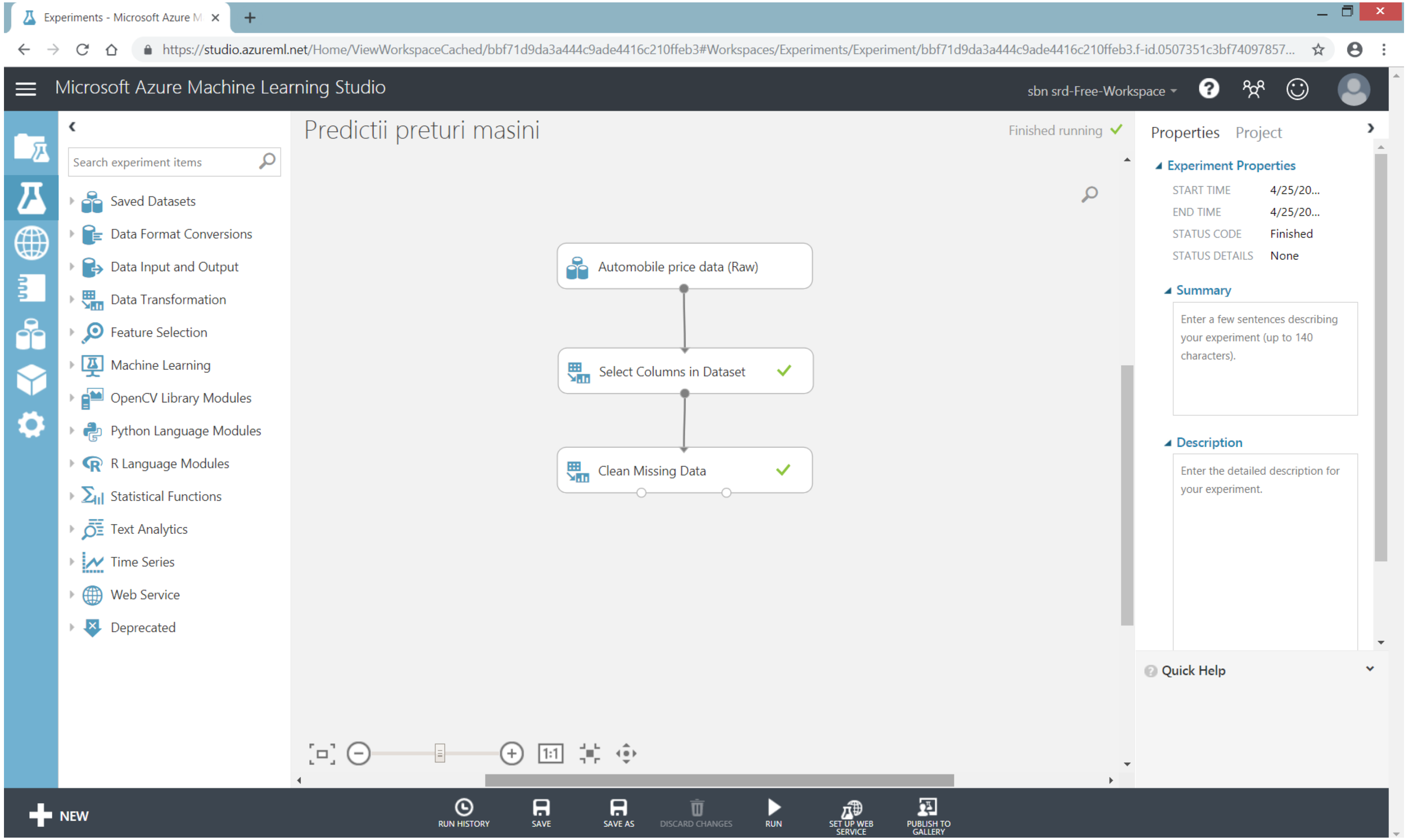
- *Clean Missing Data*
  - *Cleaning mode*
    - *Remove entire row*



Car Price Prediction

* running the experiment

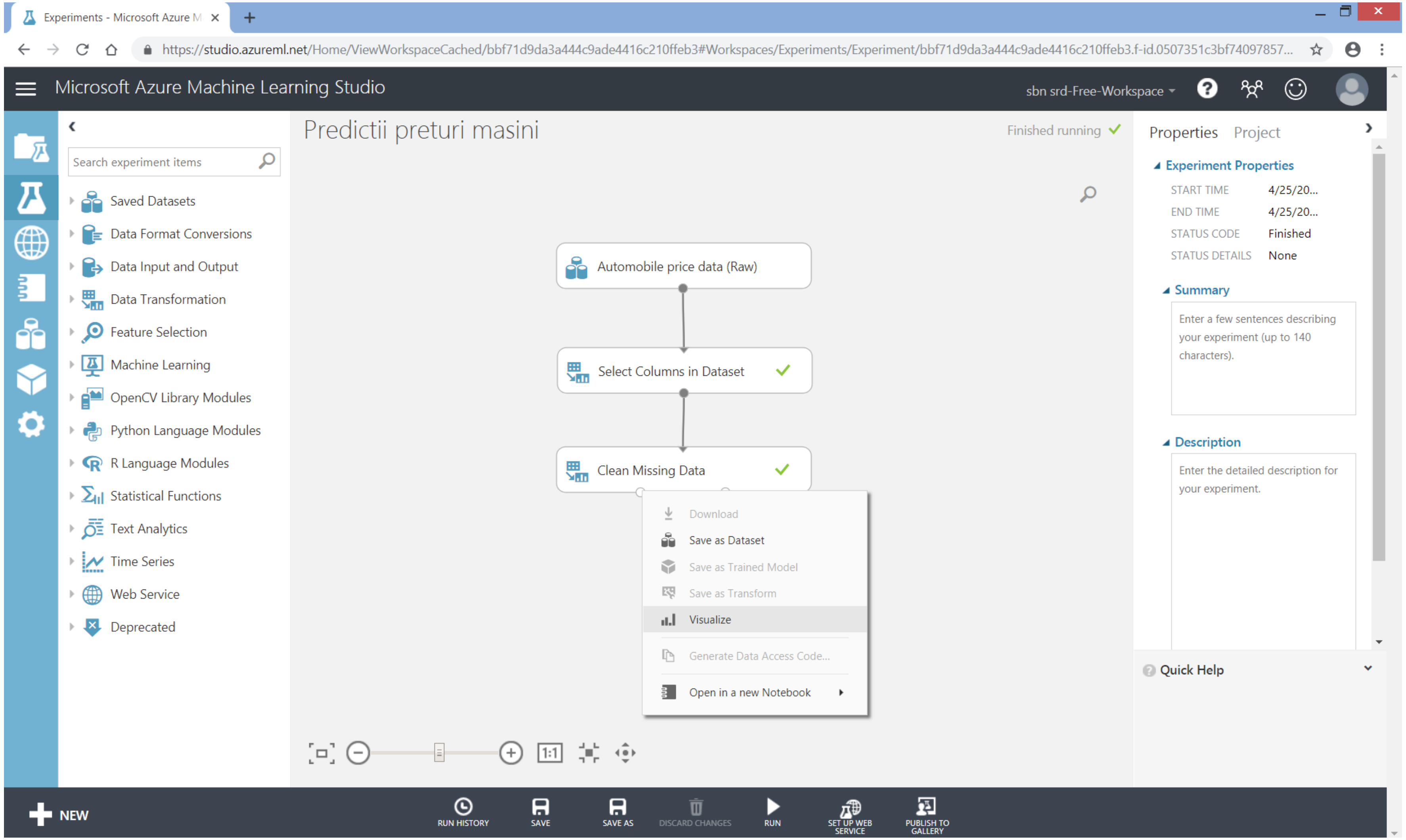
- *Run*



Car Price Prediction

* displaying the data

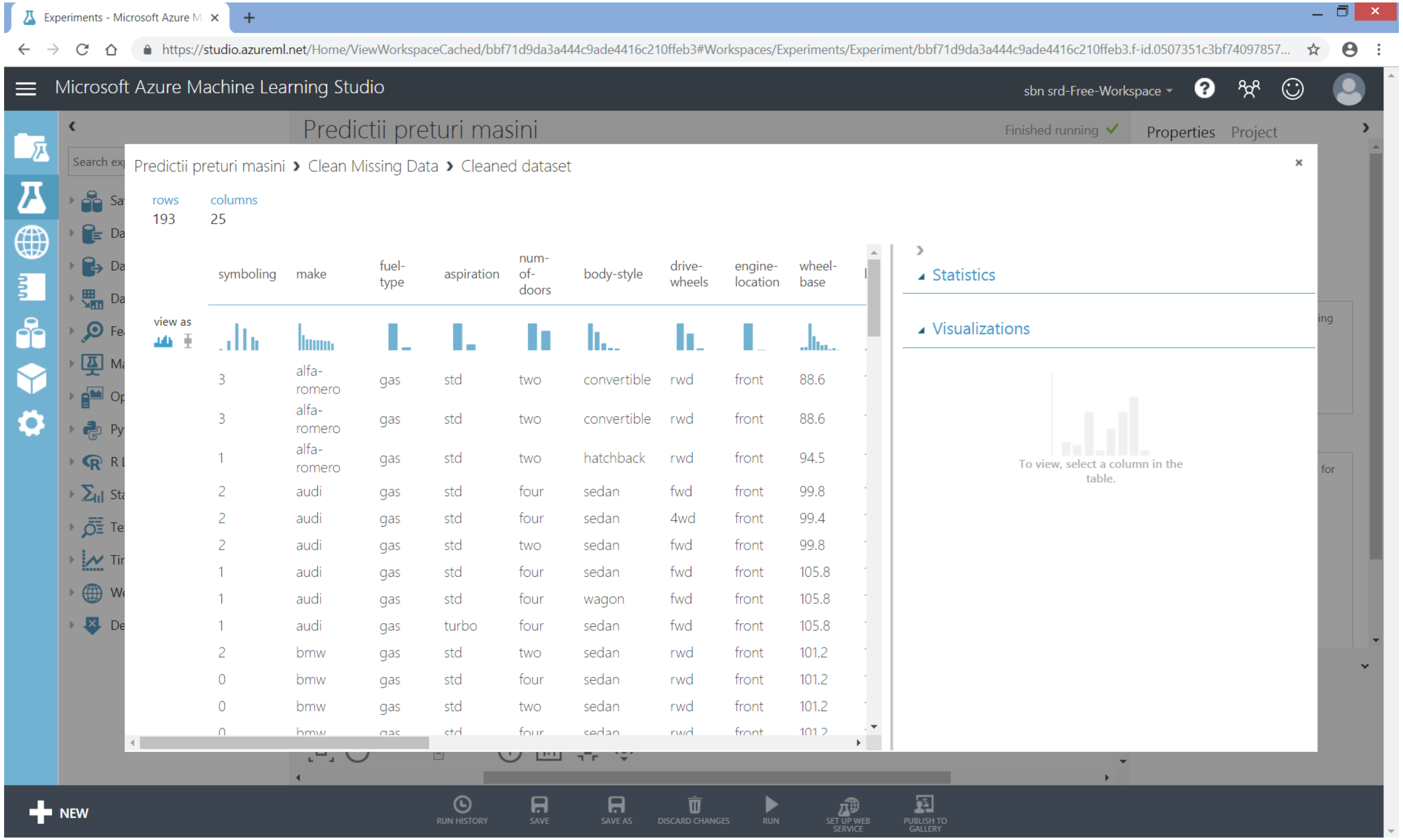
- *Clean Missing Data* module -> left output port -> *Visualize*



Car Price Prediction

* displaying the data

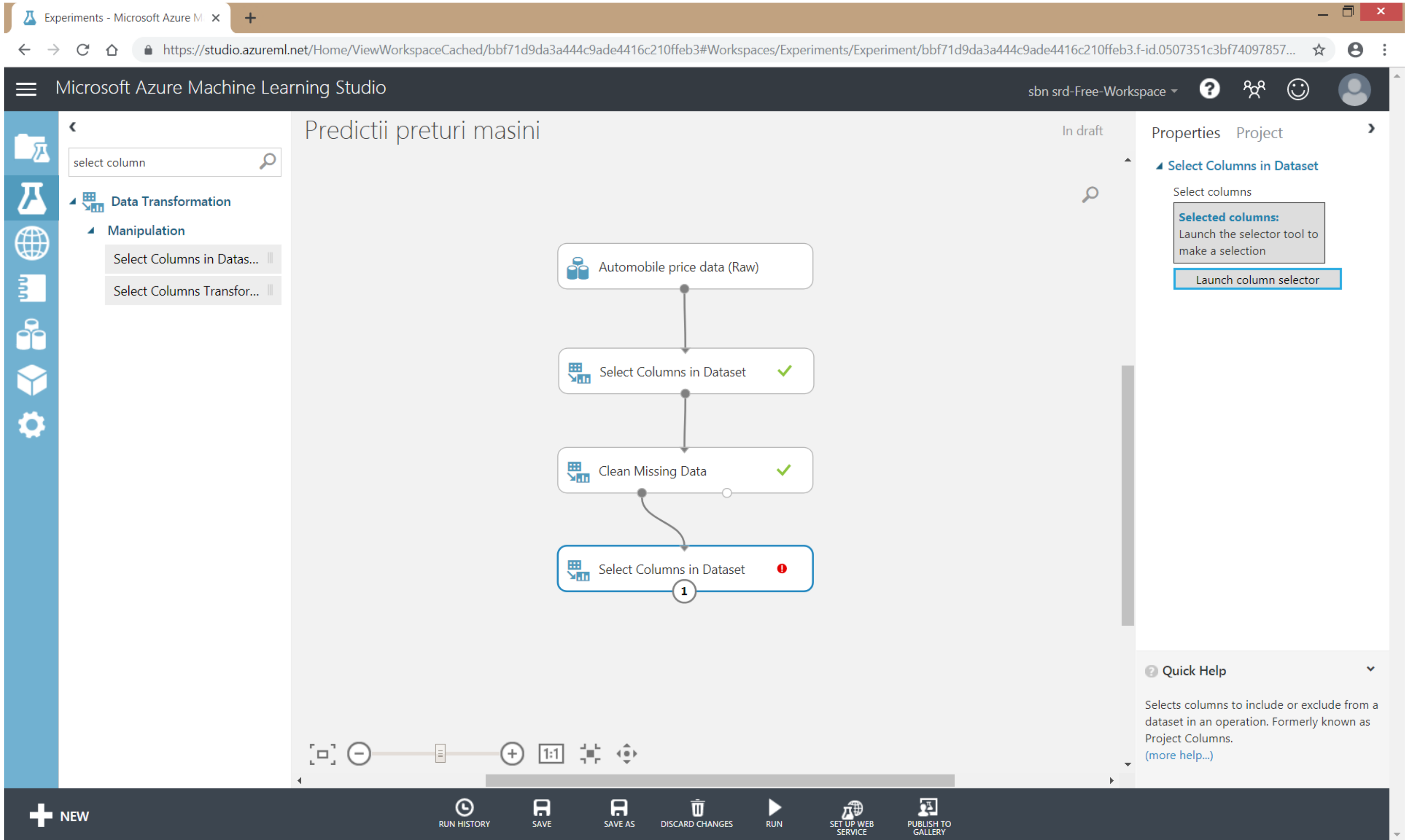
- *Clean Missing Data* module -> left output port -> *Visualize*



Car Price Prediction

* defining the *features*

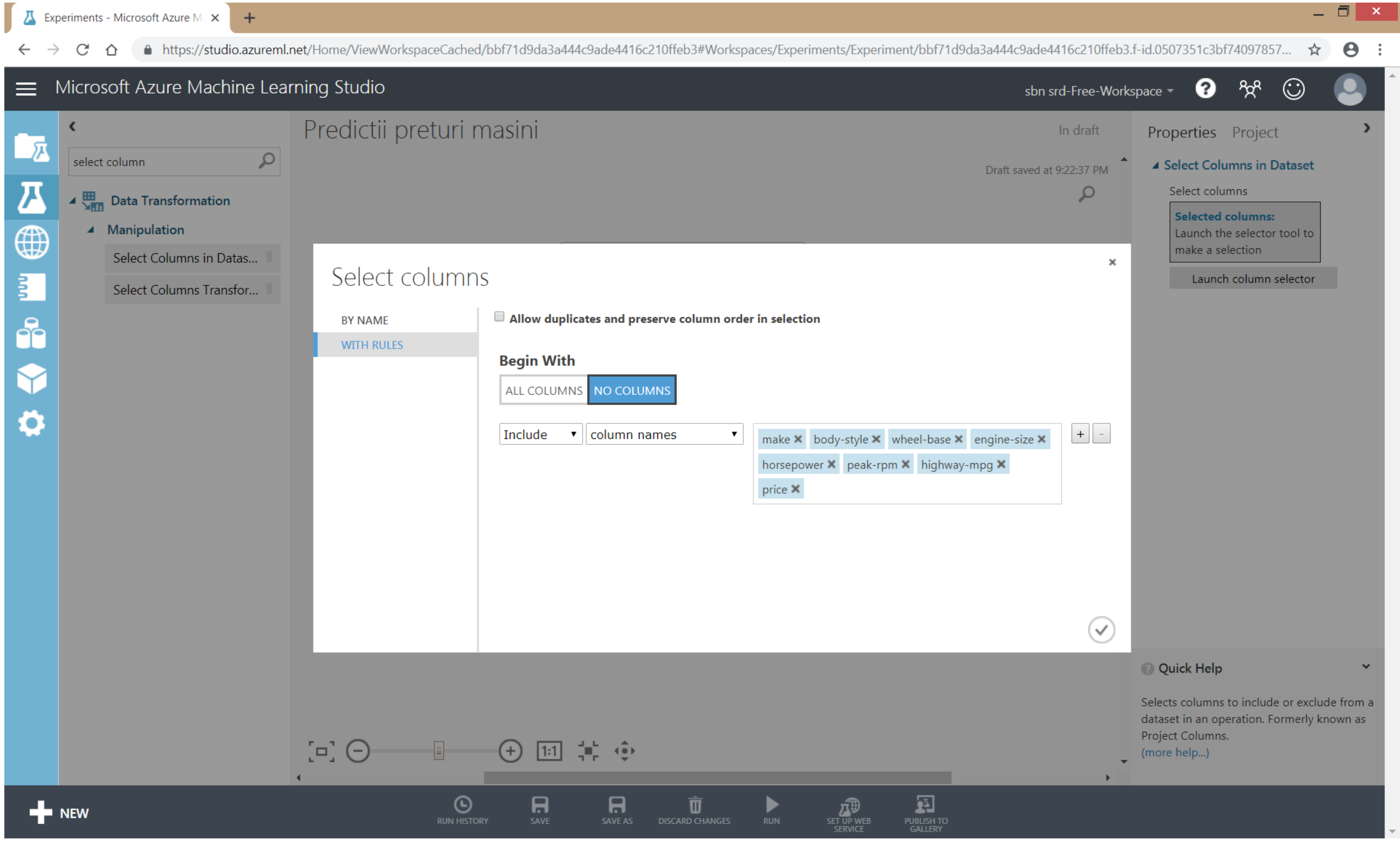
- used to create the predictive model
- *Select Columns in Dataset* module



Car Price Prediction

* defining the *features*

- *Select Columns in Dataset*
  - *Launch column selector*
    - *Begin With*
      - *No columns*
    - *Include*
      - *make, body-style, wheel-base, engine-size, horsepower, peak-rpm, highway-mpg, price*

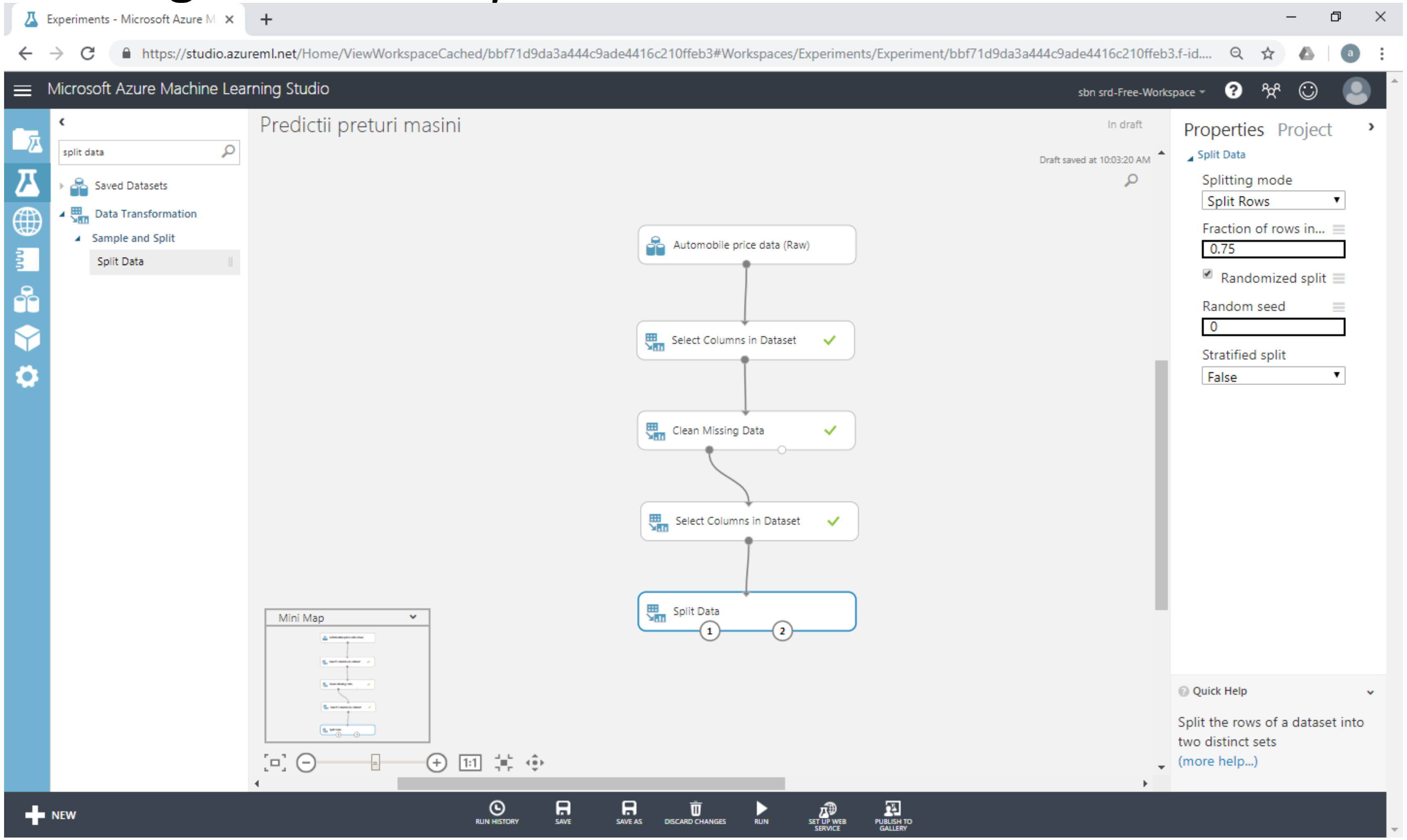


- goal: predict car price from selected features

Car Price Prediction

* choosing / applying the algorithm

- create the training / testing datasets - *Split Data* module



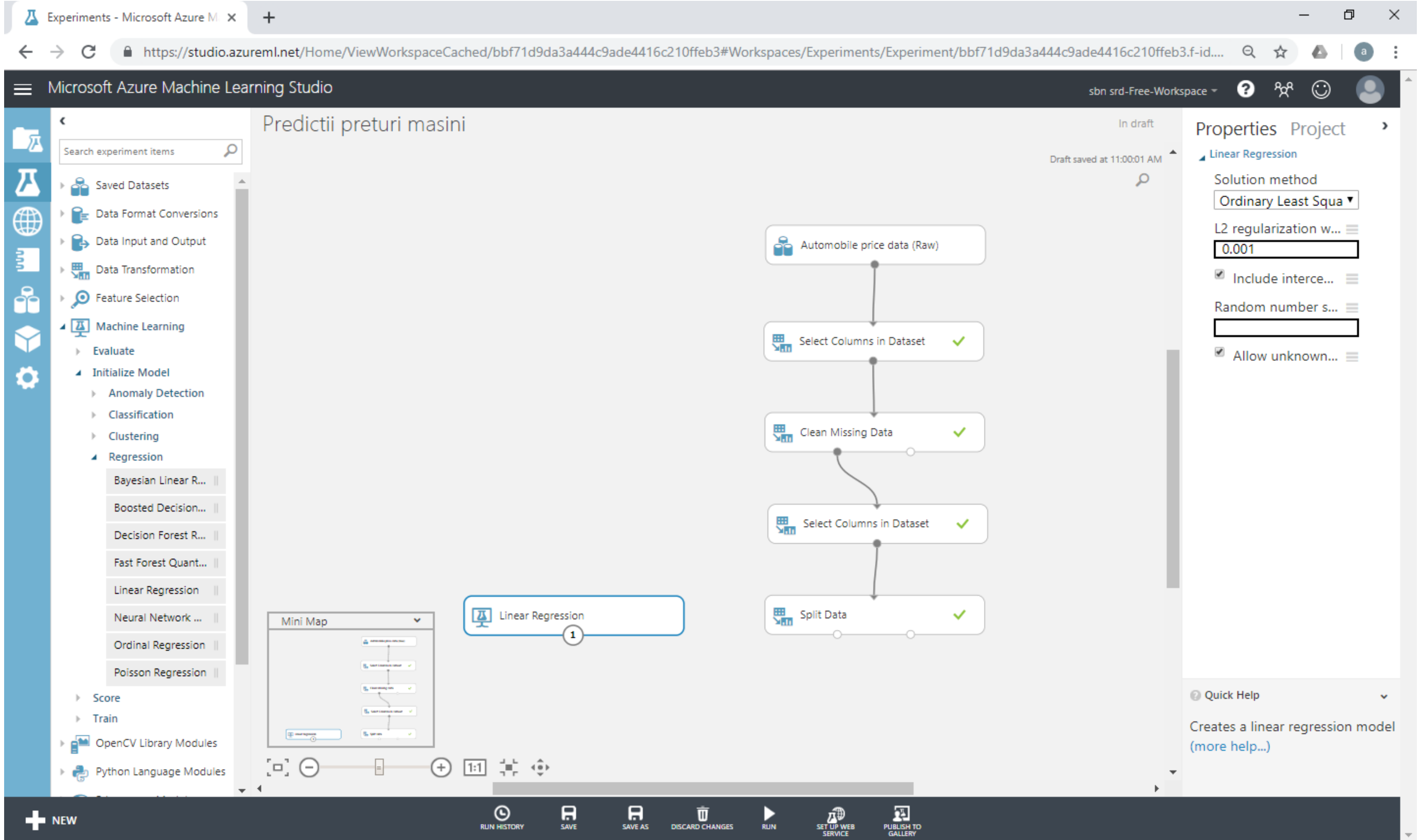
- *Split Data*
  - *Fraction of rows in the first output dataset*
    - *0.75*
    - i.e., training dataset - 75% of the data

- *Run* experiment

Car Price Prediction

* choosing / applying the algorithm

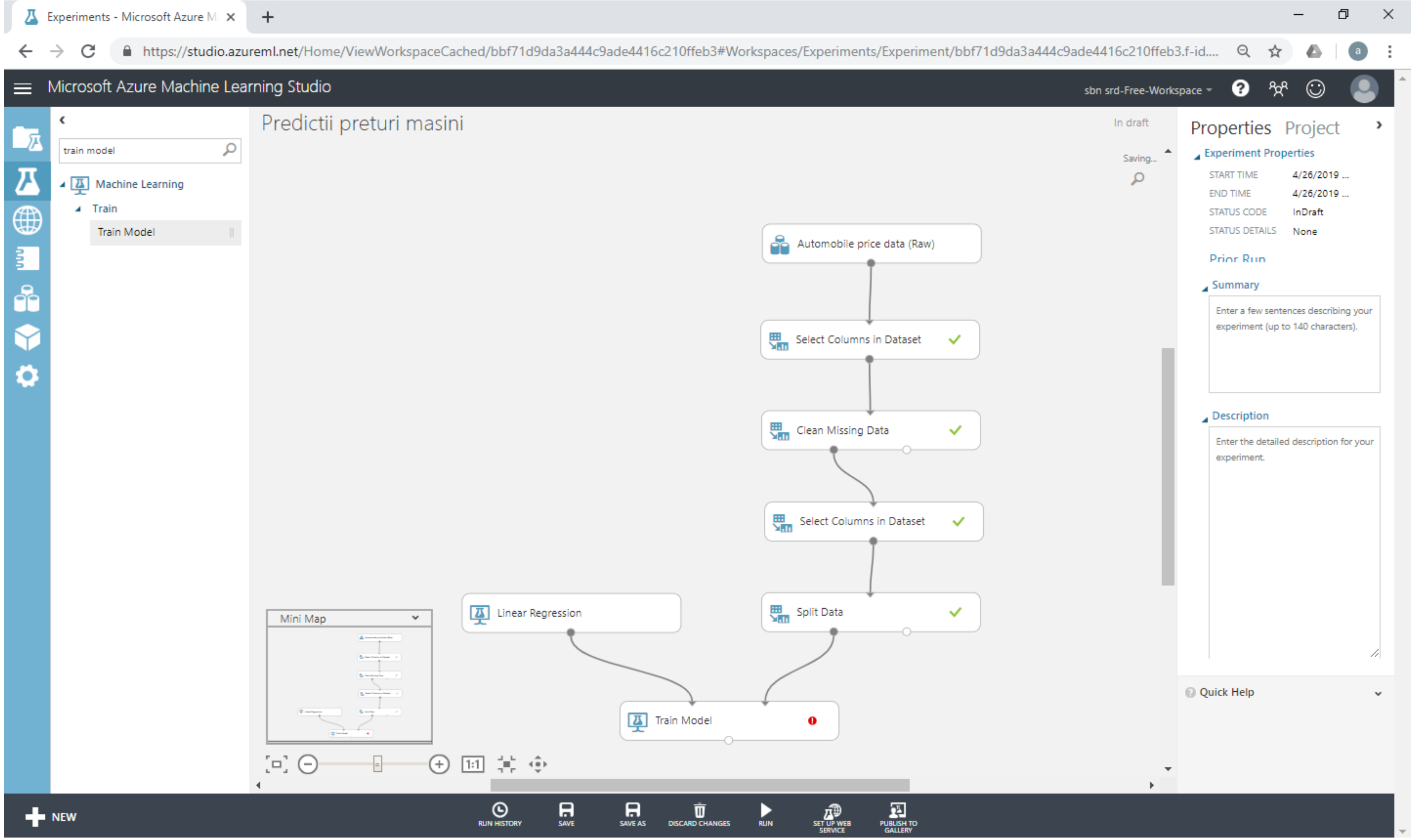
- *Machine Learning -> Initialize Model -> Regression -> Linear Regression*



Car Price Prediction

* choosing / applying the algorithm

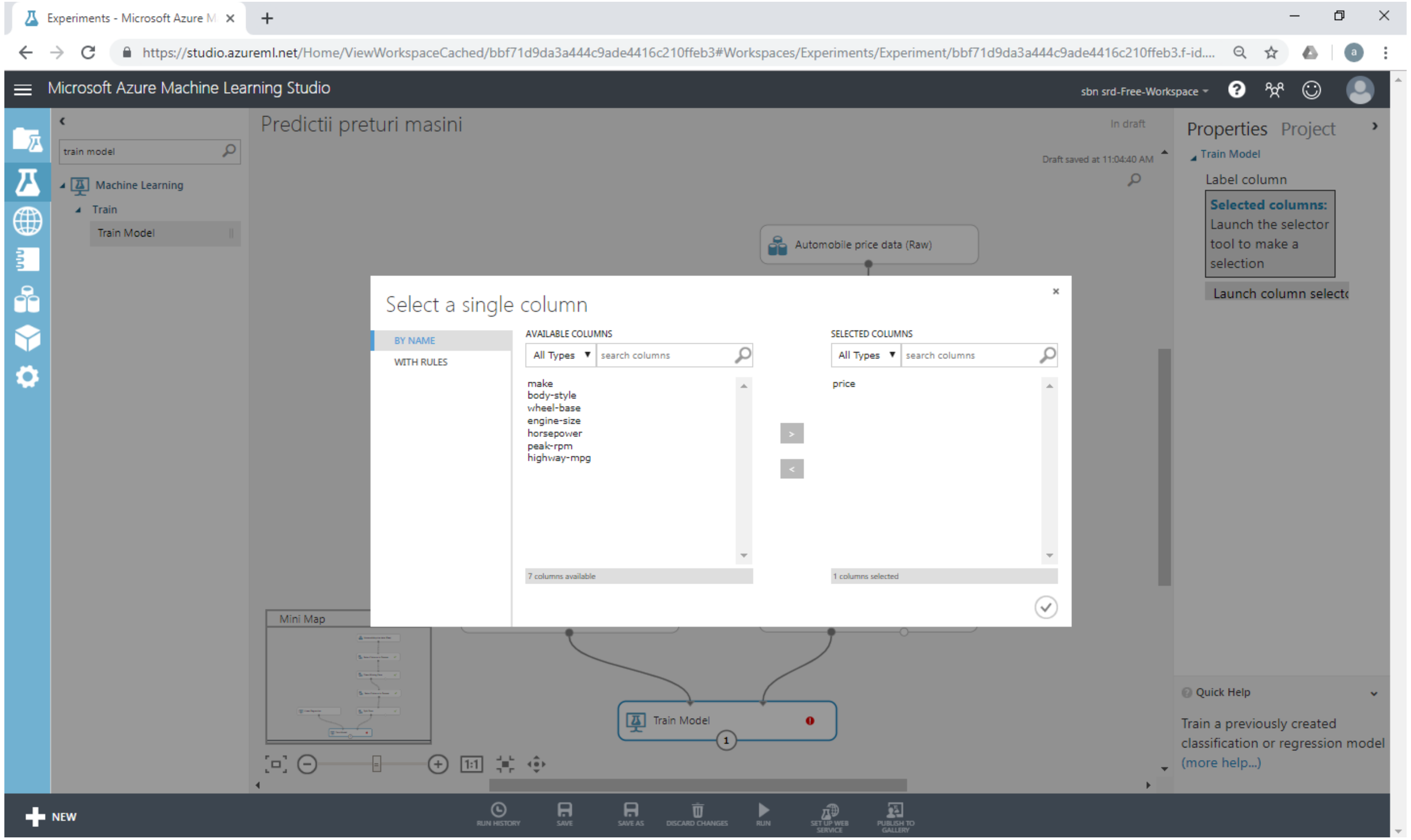
- *Train Model* module



Car Price Prediction

* choosing / applying the algorithm

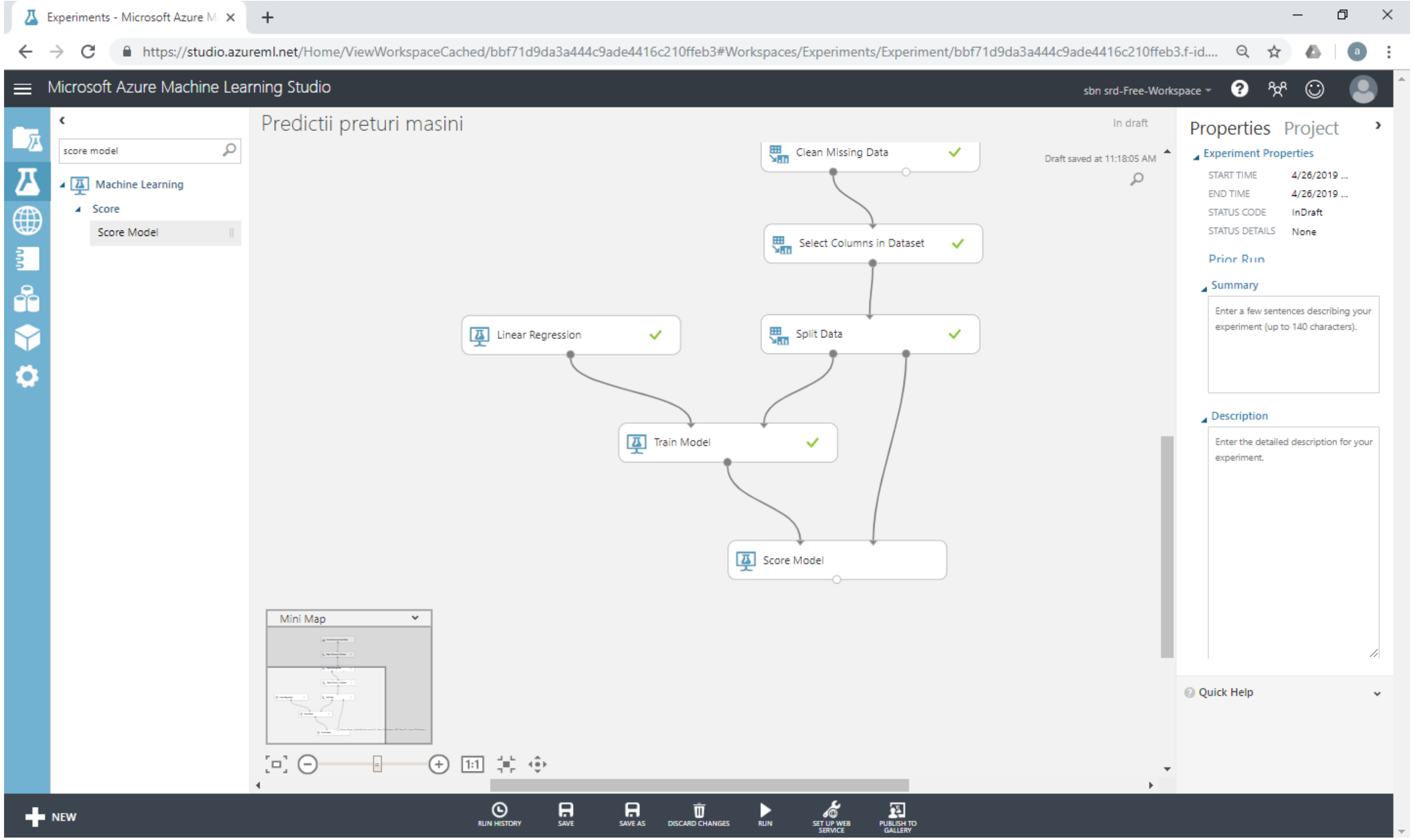
- *Train Model*
  - *Launch column selector*
    - move column *price* from *Available columns* to *Selected columns*
- *Run* experiment



Car Price Prediction

* testing the model - *Score Model* module

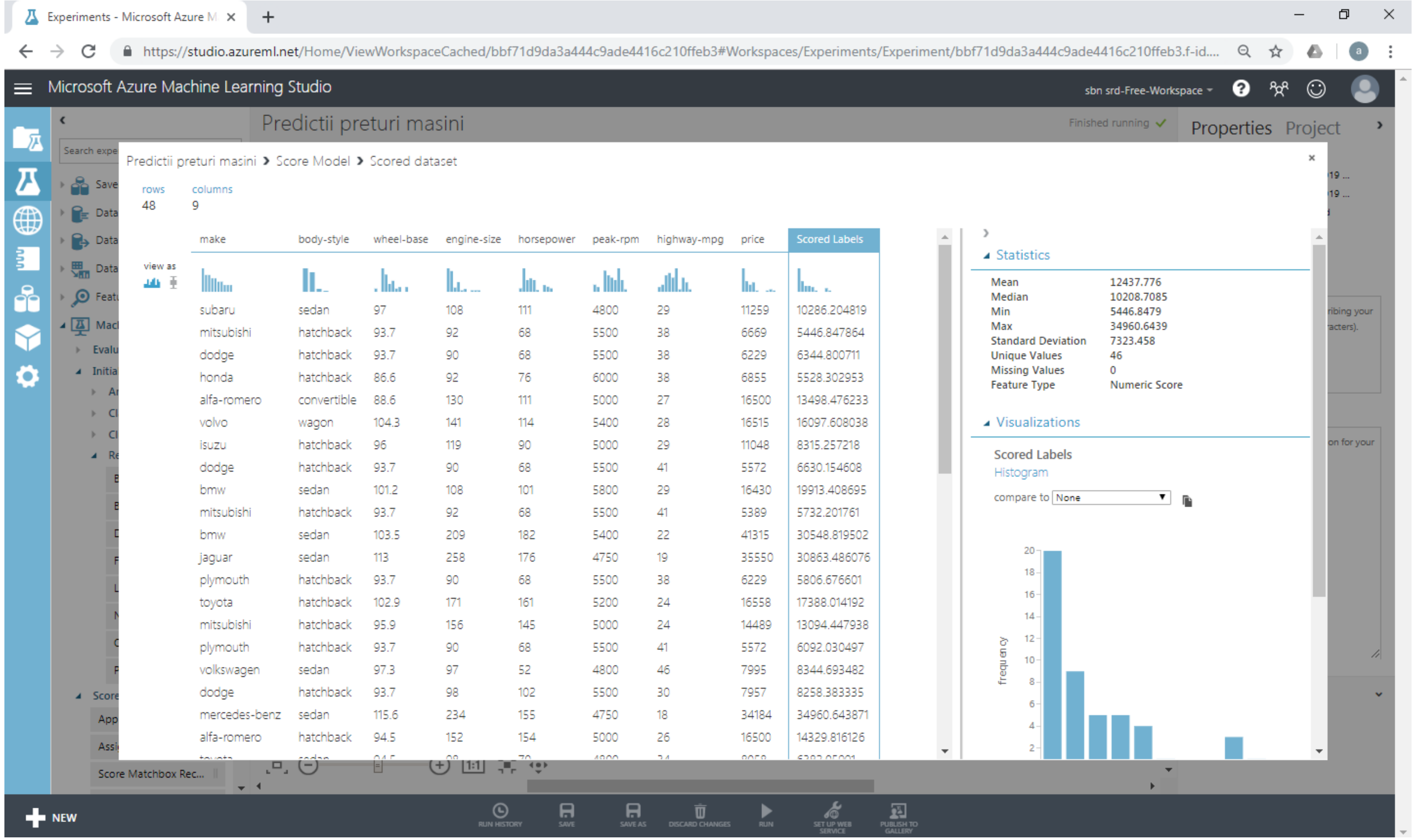
- *Run* experiment



Car Price Prediction

* testing the model

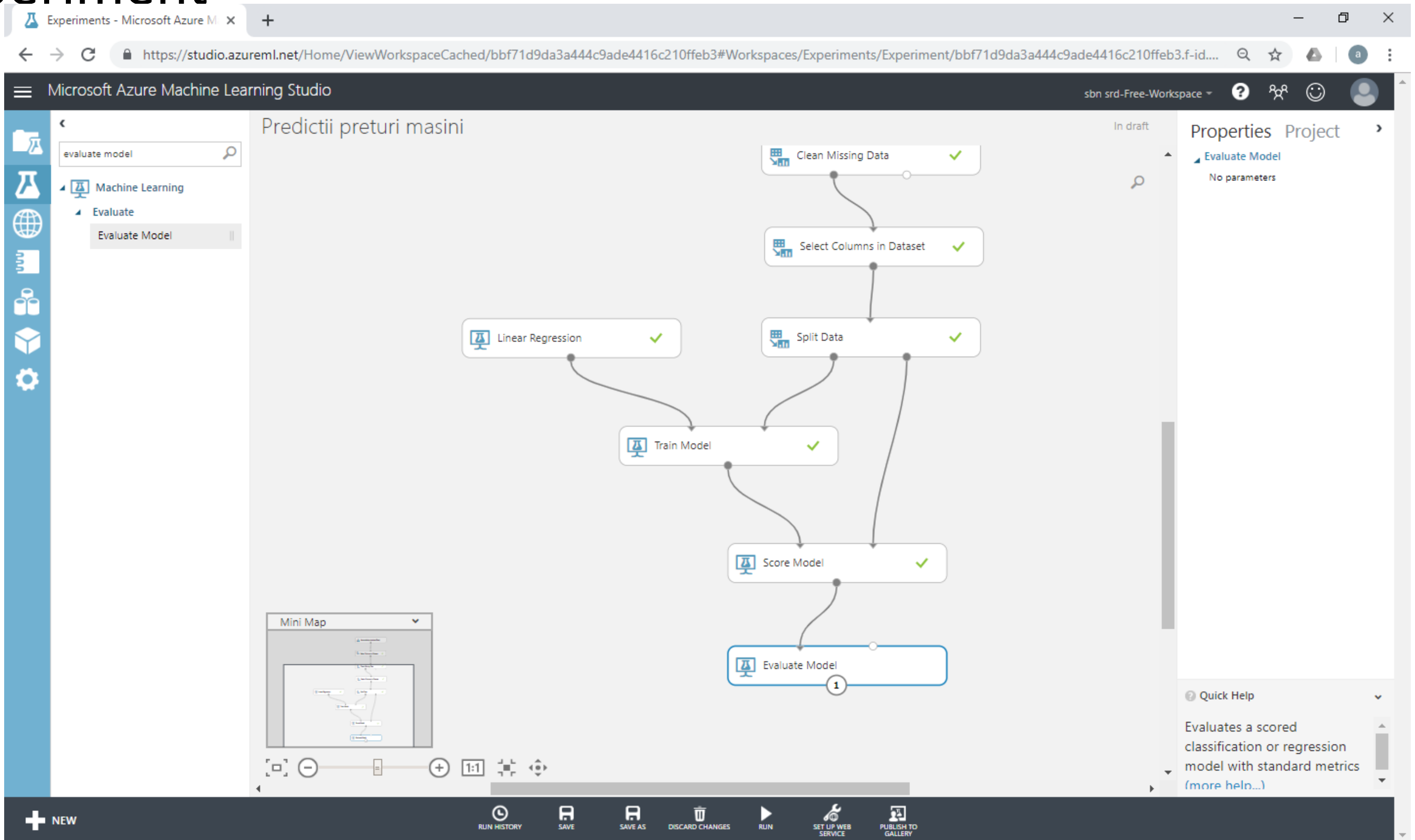
- *Score Model* output port-> *Visualize*
    - estimated / actual values for the *price* column



Car Price Prediction

* testing the model

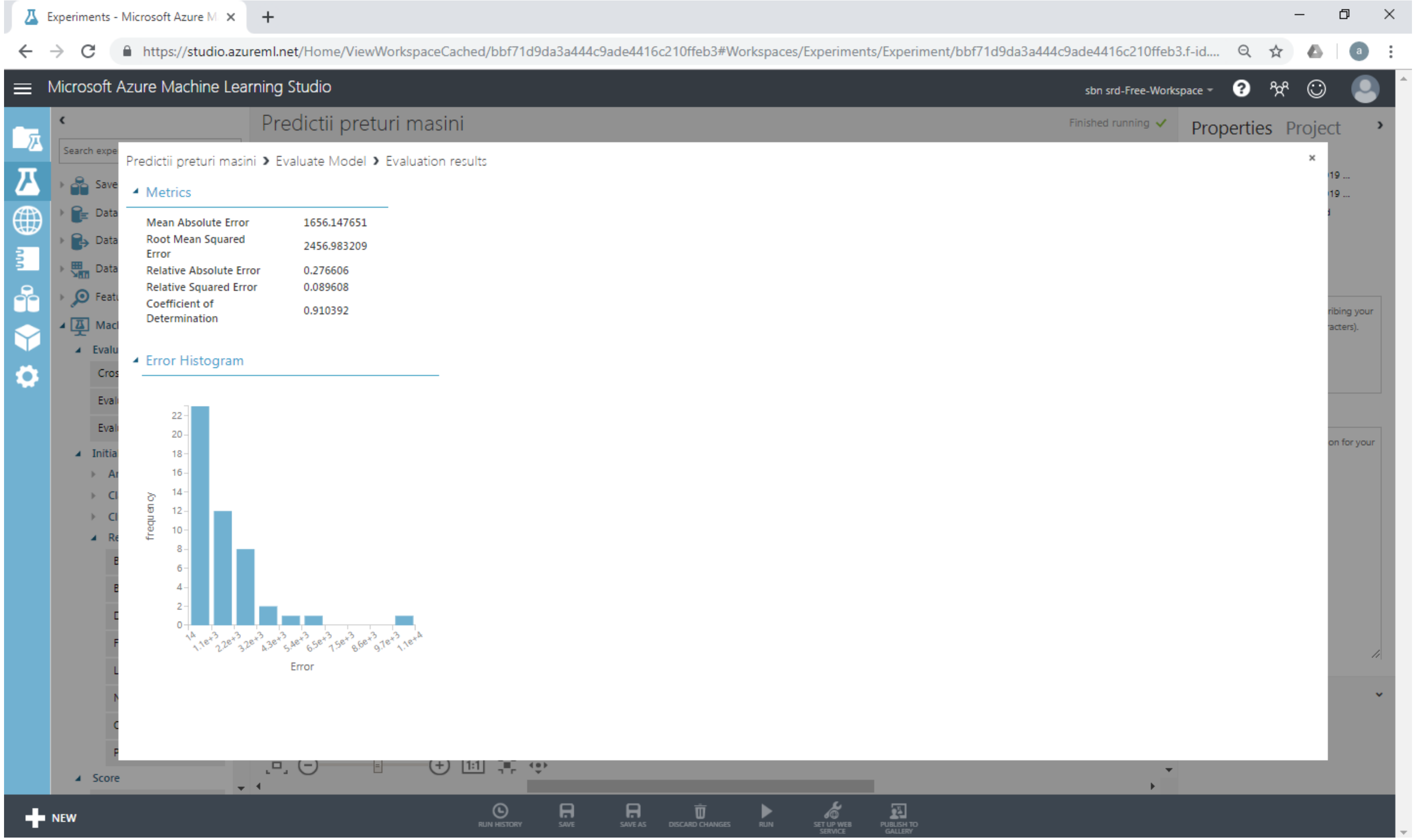
- *Evaluate Model* module
- *Run* experiment



Car Price Prediction

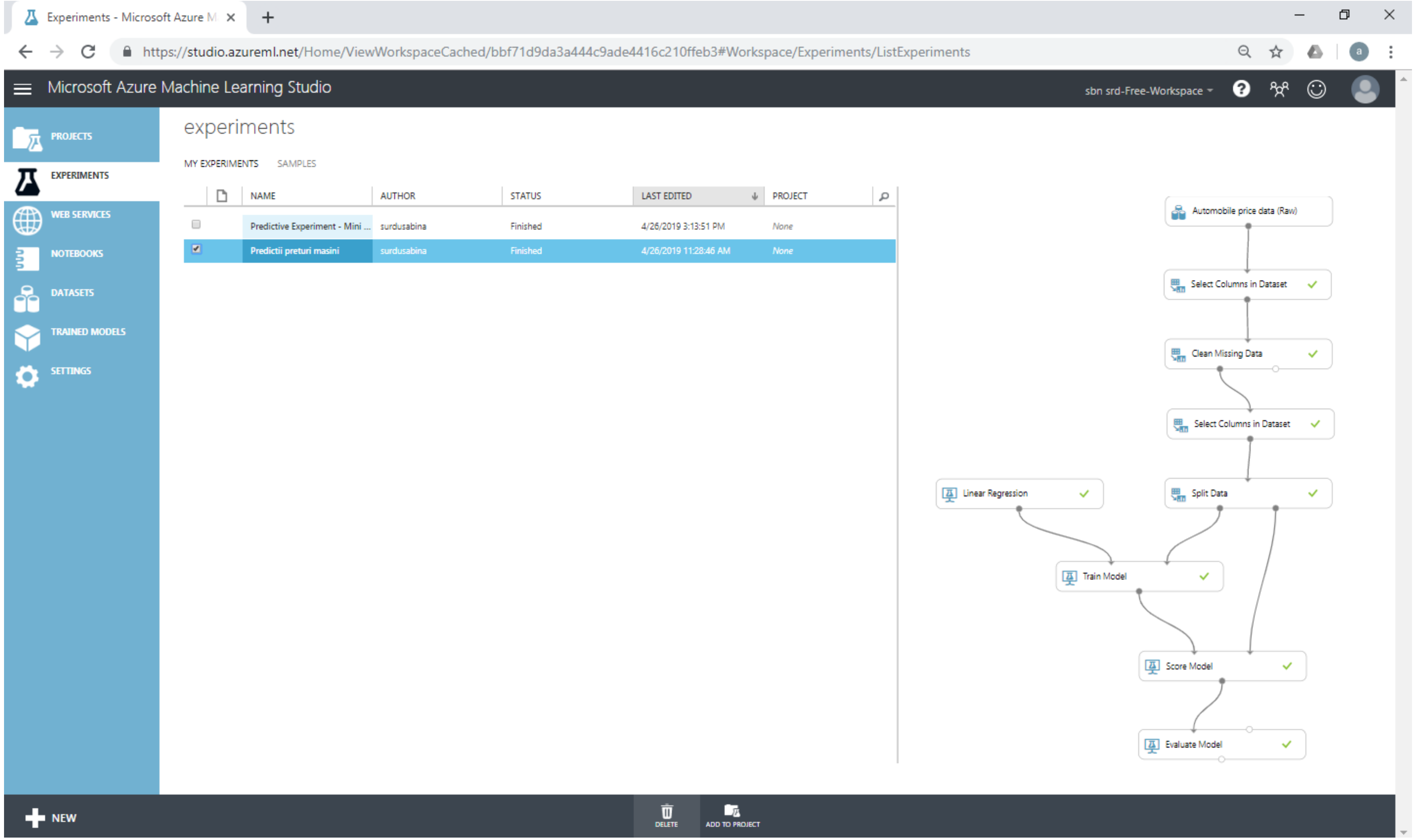
* testing the model

- *Evaluate Model* output port -> *Visualize*



Car Price Prediction

* eliminate resources



Car Price Prediction

* eliminate resources

- delete workspace: *Settings -> Delete*