TLC Manual

Table of Contents

[Introduction 13](#_Toc401580967)

[Classification 13](#_Toc401580968)

[Outputs vs. probabilities 13](#_Toc401580969)

[Multi-class classification techniques 14](#_Toc401580970)

[Measuring the quality of the classifier 15](#_Toc401580971)

[Regression 16](#_Toc401580972)

[Measuring the quality of the regressor 16](#_Toc401580973)

[Quantile Regression 17](#_Toc401580974)

[Anomaly Detection 18](#_Toc401580975)

[Detection rate at k false positives 18](#_Toc401580976)

[Detection rate at fraction p false positives 18](#_Toc401580977)

[Detection rate at number of anomalies 18](#_Toc401580978)

[AUC 18](#_Toc401580979)

[Ranking 18](#_Toc401580980)

[Measuring the quality of the ranker 19](#_Toc401580981)

[Getting started 21](#_Toc401580982)

[Input data format 23](#_Toc401580983)

[Binary input format (TLCBIN) 24](#_Toc401580984)

[Label Scale 24](#_Toc401580985)

[Feature Scale 24](#_Toc401580986)

[Buffer Instance Count 25](#_Toc401580987)

[Threads 25](#_Toc401580988)

[Artificial group size 25](#_Toc401580989)

[Weights 25](#_Toc401580990)

[Names 26](#_Toc401580991)

[Attributes 26](#_Toc401580992)

[Text input format 26](#_Toc401580993)

[Has Header 29](#_Toc401580994)

[Allow Sparse 30](#_Toc401580995)

[Categorical Features, Text Features, Categorical Handler, Text Handler 30](#_Toc401580996)

[Separator 39](#_Toc401580997)

[Feature exclusion and inclusion 40](#_Toc401580998)

[Header Filename 41](#_Toc401580999)

[Metadata: names and attributes 41](#_Toc401581000)

[Label column Idx 42](#_Toc401581001)

[No Label 42](#_Toc401581002)

[Weight column Idx 42](#_Toc401581003)

[Label Parser 43](#_Toc401581004)

[Log Features and Log Odds Features 43](#_Toc401581005)

[Sparse and dense formats, Max Dimensionality 44](#_Toc401581006)

[Accept Unlabeled 45](#_Toc401581007)

[Trim whitespace 46](#_Toc401581008)

[Blank Missing Values 46](#_Toc401581009)

[Instance Filters 46](#_Toc401581010)

[Max Num Instances 47](#_Toc401581011)

[Max Num Bad Instances 47](#_Toc401581012)

[Use Threads 47](#_Toc401581013)

[Example 47](#_Toc401581014)

[Missing features 48](#_Toc401581015)

[Ranking Instances 48](#_Toc401581016)

[Base Instances Class 49](#_Toc401581017)

[Group Key 49](#_Toc401581018)

[Sample Instances 50](#_Toc401581019)

[Sample Rate 50](#_Toc401581020)

[Seed 50](#_Toc401581021)

[Base Instances Class 50](#_Toc401581022)

[Bing/OSD format 51](#_Toc401581023)

[Bing Extraction format 51](#_Toc401581024)

[Label Mapping 52](#_Toc401581025)

[Smallest Positive 53](#_Toc401581026)

[Image input format 53](#_Toc401581027)

[Buffer Instance Count 54](#_Toc401581028)

[Image instance parser and settings 54](#_Toc401581029)

[Transformed Feature Instances 56](#_Toc401581030)

[Base Instances Class 56](#_Toc401581031)

[Transformer 56](#_Toc401581032)

[Example 57](#_Toc401581033)

[Sample datasets 57](#_Toc401581034)

[Output Model Formats 61](#_Toc401581035)

[TLC format (binary) 61](#_Toc401581036)

[.ini format 61](#_Toc401581037)

[Code format 64](#_Toc401581038)

[Text format 64](#_Toc401581039)

[Model summary 66](#_Toc401581040)

[Specifying the learning mode 69](#_Toc401581041)

[Cross validation mode 69](#_Toc401581042)

[Train Dataset 70](#_Toc401581043)

[Number of Folds 70](#_Toc401581044)

[Instances Type 71](#_Toc401581045)

[Cache Instances in Memory 71](#_Toc401581046)

[Stratify Folds, Stratification Key, Stratify Folds Sequentially 71](#_Toc401581047)

[Train Set Fraction 72](#_Toc401581048)

[Validation Dataset 72](#_Toc401581049)

[Evaluator Class 73](#_Toc401581050)

[Number of Runs 73](#_Toc401581051)

[Azure Storage Connection String 73](#_Toc401581052)

[Example 74](#_Toc401581053)

[Train-test mode 75](#_Toc401581054)

[Train Dataset 76](#_Toc401581055)

[Test Dataset 76](#_Toc401581056)

[Instances Type 76](#_Toc401581057)

[Cache Instances in Memory 76](#_Toc401581058)

[Trained Model Folder 77](#_Toc401581059)

[Save Model Summary, Save Model as Text/Ini/Code 77](#_Toc401581060)

[Train Set Fraction 77](#_Toc401581061)

[Bootstrap rounds, Bootstrap test set 78](#_Toc401581062)

[Validation Dataset 78](#_Toc401581063)

[Evaluator Class 78](#_Toc401581064)

[Azure Storage Connection String 79](#_Toc401581065)

[Test Instance Settings 79](#_Toc401581066)

[Example 79](#_Toc401581067)

[Train mode 80](#_Toc401581068)

[Trained Model Folder 81](#_Toc401581069)

[Train Dataset 81](#_Toc401581070)

[Instances Type 82](#_Toc401581071)

[Cache Instances in Memory 82](#_Toc401581072)

[Save Model Summary, Save Model as Text/Ini/Code 82](#_Toc401581073)

[Train Set Fraction 83](#_Toc401581074)

[Validation Dataset 83](#_Toc401581075)

[Azure Storage Connection String 83](#_Toc401581076)

[Example 84](#_Toc401581077)

[Test mode 85](#_Toc401581078)

[Trained Model 85](#_Toc401581079)

[Test Dataset 86](#_Toc401581080)

[Instances Type 86](#_Toc401581081)

[Predict Only 86](#_Toc401581082)

[Evaluator Class 86](#_Toc401581083)

[Azure Storage Connection String 87](#_Toc401581084)

[Example 87](#_Toc401581085)

[Other TLC modes 91](#_Toc401581086)

[CreateInstances mode 91](#_Toc401581087)

[Converting data into standard text format 91](#_Toc401581088)

[Converting data into binary format 92](#_Toc401581089)

[Data normalization 92](#_Toc401581090)

[Per-feature statistics 93](#_Toc401581091)

[FeatureSelection mode 93](#_Toc401581092)

[/fsm 93](#_Toc401581093)

[/fso 93](#_Toc401581094)

[/fsf 93](#_Toc401581095)

[/fsa 94](#_Toc401581096)

[Examples 95](#_Toc401581097)

[Other alternatives to FeatureSelection mode 96](#_Toc401581098)

[ML Basics 97](#_Toc401581099)

[Choosing the right algorithm 97](#_Toc401581100)

[Examples 97](#_Toc401581101)

[Feature normalization 99](#_Toc401581102)

[Min-Max normalizer 100](#_Toc401581103)

[Gaussian normalizer 101](#_Toc401581104)

[Binning normalizer 101](#_Toc401581105)

[Feature transformation 101](#_Toc401581106)

[Random Fourier Features Transformer (RFF Transformer) 102](#_Toc401581107)

[PCA Projection Transformer 104](#_Toc401581108)

[Regularization 106](#_Toc401581109)

[Using parameter sweeps 107](#_Toc401581110)

[Using Parameter Sweeps from the Command Line 110](#_Toc401581111)

[TLC Learners 115](#_Toc401581112)

[Logistic regression 117](#_Toc401581113)

[Overview 117](#_Toc401581114)

[Settings 117](#_Toc401581115)

[Weighting examples 121](#_Toc401581116)

[Neural networks 121](#_Toc401581117)

[Overview 121](#_Toc401581118)

[Settings 122](#_Toc401581119)

[Boosted trees 132](#_Toc401581120)

[Averaged perceptron 132](#_Toc401581121)

[Overview 132](#_Toc401581122)

[Settings 133](#_Toc401581123)

[SVM 139](#_Toc401581124)

[Overview 139](#_Toc401581125)

[Settings 140](#_Toc401581126)

[Random forests (Bramble) 144](#_Toc401581127)

[Vowpal Wabbit 145](#_Toc401581128)

[FastTree 145](#_Toc401581129)

[Overview 145](#_Toc401581130)

[Settings 146](#_Toc401581131)

[Weighting examples 152](#_Toc401581132)

[Parallel Ensemble 152](#_Toc401581133)

[Overview 152](#_Toc401581134)

[Settings 153](#_Toc401581135)

[Local Deep SVM 161](#_Toc401581136)

[Overview 161](#_Toc401581137)

[Settings 162](#_Toc401581138)

[Sample Results 165](#_Toc401581139)

[StratoLearner 169](#_Toc401581140)

[Overview 169](#_Toc401581141)

[Settings 169](#_Toc401581142)

[One-vs-all / pairwise coupling 171](#_Toc401581143)

[Overview 171](#_Toc401581144)

[Settings 171](#_Toc401581145)

[Multi-class image patching 173](#_Toc401581146)

[Overview 173](#_Toc401581147)

[Settings 174](#_Toc401581148)

[Poisson regression 176](#_Toc401581149)

[Overview 176](#_Toc401581150)

[Settings 177](#_Toc401581151)

[Stochastic gradient descent 180](#_Toc401581152)

[Overview 180](#_Toc401581153)

[Settings 180](#_Toc401581154)

[One Class SVM 185](#_Toc401581155)

[Overview 185](#_Toc401581156)

[Settings 186](#_Toc401581157)

[Fast Forest 188](#_Toc401581158)

[Overview 188](#_Toc401581159)

[Settings 188](#_Toc401581160)

[Quantile Regression Tester 191](#_Toc401581161)

[PCA Anomaly Detector 192](#_Toc401581162)

[Overview 192](#_Toc401581163)

[Settings 192](#_Toc401581164)

[K-means++ 194](#_Toc401581165)

[Overview 194](#_Toc401581166)

[Settings 194](#_Toc401581167)

[Fast Linear Classifier 196](#_Toc401581168)

[Overview 196](#_Toc401581169)

[Settings 196](#_Toc401581170)

[Time Series Anomaly Detector 200](#_Toc401581171)

[Sliding Window Data 200](#_Toc401581172)

[Detect Anomalies 201](#_Toc401581173)

[Deep Structured Semantic Modeling 201](#_Toc401581174)

[Overview 201](#_Toc401581175)

[Settings 201](#_Toc401581176)

[How to use DSSM from the command line 204](#_Toc401581177)

[Choosing the output options 207](#_Toc401581178)

[Print Summary 207](#_Toc401581179)

[Debug Level 207](#_Toc401581180)

[Output folder 207](#_Toc401581181)

[Save Output Summary 208](#_Toc401581182)

[Save per-Instance Results 208](#_Toc401581183)

[Save PR File 209](#_Toc401581184)

[Choosing the run options 211](#_Toc401581185)

[Run mode 211](#_Toc401581186)

[Run distributed on HPC cluster 211](#_Toc401581187)

[TL.exe Path 212](#_Toc401581188)

[Random Seed 212](#_Toc401581189)

[Use Threads 212](#_Toc401581190)

[Max concurrent processes 212](#_Toc401581191)

[Extra Assemblies 212](#_Toc401581192)

[Running Experiments 213](#_Toc401581193)

[Running experiments on the command line 213](#_Toc401581194)

[Running experiments on TLC GUI 215](#_Toc401581195)

[Numerical results and comparison 215](#_Toc401581196)

[Keyboard shortcuts in TLC GUI 216](#_Toc401581197)

[ResultProcessor 217](#_Toc401581198)

[TLC with SCOPE 219](#_Toc401581199)

[TLC Learner Scope Module 219](#_Toc401581200)

[TLC Scope Module Scripts 221](#_Toc401581201)

[TLC Scope Module Deployment 221](#_Toc401581202)

[TLC Scope Large-Scale Train and Prediction Scripts 221](#_Toc401581203)

[Utility and Etc Scripts 221](#_Toc401581204)

[Bug Fixes 221](#_Toc401581205)

[Aether: Learning via Aether modules 223](#_Toc401581206)

[Understanding the model 225](#_Toc401581207)

[Tree Visualizer 225](#_Toc401581208)

[Visualizing without data 226](#_Toc401581209)

[Visualizing with data 226](#_Toc401581210)

[Instances Only layout 227](#_Toc401581211)

[Show instances on Grid 228](#_Toc401581212)

[Selecting instances on Grid 228](#_Toc401581213)

[ErrorExplorer: support for improving accuracy 229](#_Toc401581214)

[Net# (CloudML Neural Network Definition Language) 231](#_Toc401581215)

[Basic Concepts 231](#_Toc401581216)

[Examples 231](#_Toc401581217)

[Hello World Example 231](#_Toc401581218)

[Sophomore Example 232](#_Toc401581219)

[Digit Recognition Example 232](#_Toc401581220)

[Layers 233](#_Toc401581221)

[Type System 234](#_Toc401581222)

[Values and Constants 234](#_Toc401581223)

[Default Structure 235](#_Toc401581224)

[Language Basics 235](#_Toc401581225)

[Structure Specification 237](#_Toc401581226)

[Net 237](#_Toc401581227)

[Constant Declarations 238](#_Toc401581228)

[Layer Declarations 238](#_Toc401581229)

[Share Declaration 247](#_Toc401581230)

[Expressions 248](#_Toc401581231)

[Boolean operators 248](#_Toc401581232)

[Comparison operators 248](#_Toc401581233)

[Concatenation operator 250](#_Toc401581234)

[Addition 250](#_Toc401581235)

[Multiplication 250](#_Toc401581236)

[Prefix unary operators 250](#_Toc401581237)

[Exponentiation 250](#_Toc401581238)

[Postfix operators 251](#_Toc401581239)

[Primary expressions 251](#_Toc401581240)

[Invocation 252](#_Toc401581241)

[TLC API Tutorial 253](#_Toc401581242)

[Setup 253](#_Toc401581243)

[Using existing trainers and predictors 254](#_Toc401581244)

[Building a console application that trains and saves a model 254](#_Toc401581245)

[Loading and using trained models to predict 256](#_Toc401581246)

[Predicting probabilities 258](#_Toc401581247)

[Predicting for multi-class 258](#_Toc401581248)

[Choosing the learner based on user input 259](#_Toc401581249)

[Using non default arguments 259](#_Toc401581250)

[Using the Native API to load TLC predictors 260](#_Toc401581251)

[Creating new trainers and predictors 261](#_Toc401581252)

[Creating a predictor 261](#_Toc401581253)

[Creating a trainer 266](#_Toc401581254)

[Creating a trainer factory 268](#_Toc401581255)

[Adding SubComponent arguments 271](#_Toc401581256)

[Using SubComponents that have their own arguments 274](#_Toc401581257)

[Binary serialization of predictors 276](#_Toc401581258)

[Testing the new learner 282](#_Toc401581259)

[Index 285](#_Toc401581260)

# Introduction

TLC is a machine learning toolkit for performing large scale machine learning experiments.   It provides:

* Standard supervised learning tasks: Binary and Multi-class classification, regression, quantile regression, anomaly detection and ranking, with diverse state-of-the-art learning algorithms including Boosted Decision Trees and Neural Networks, Logistic Regression, SVM, Averaged Perceptron and Stochastic Gradient Descent regression
* Support for core experimentation scenarios: Cross-validation, Train-then-test, Training and saving models for use in production
* Running in standalone GUI mode or command line mode
* A .NET API which can be consumed by custom applications, Cosmos and Aether scripts
* TLC GUI allows running experiments as jobs on HPC clusters or on Azure
* Support for core machine learning engineering analyses (learning curves, feature selection, parameter sweeps)
* Wrappers for Bramble and VW
* Visualization and comparison of results using the TLC GUI, ResultProcessor or Tree Visualizer

## Classification

Classification algorithms are algorithms that learn to predict the class or category of an instance of data. The input of a classification algorithm is a set of labeled examples. Each example is represented as a feature vector, and each label is an integer between 0 and image002.png, where image004.pngis the number of classes. If image006.png, the task is called *binary classification*, whereas if image008.png, it is called *multi-class classification*. The output of a classification algorithm is a classifier, which can be used to predict the label of a new (unlabeled) instance.

### Outputs vs. probabilities

With the exception of Logistic Regression, which trains a classifier that outputs the probability of belonging to each class, all classifiers produce unbounded scores that can be interpreted as margins or confidences. Converting these scores to probabilities is done by training a calibration model. The probability is then converted to an actual prediction by choosing a threshold such that every probability above it is predicted as class 1, and every probability below it is predicted as class 0.

TLC provides three models for calibration: Sigmoid (or Platt calibration), Isotonic regression and naïve calibration.

#### Sigmoid calibration

This model was introduced by Platt in the paper [Probabilistic Outputs for Support Vector Machines and Comparisons to Regularized Likelihood Methods](http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.41.1639). The model finds two parameters image002.png, and outputs the probability of belonging to class 1 as

image004.png

where image006.png is the score computed for the current example, and image008.png is its predicted label. The parameters image010.png and image012.png are found by minimizing the negative log-likelihood of the training data.

#### ­Isotonic regression

This model was introduced by Zadrozni and Elkan in the papers [Obtaining Calibrated Probability Estimates From Decision Trees and Naïve Bayesian Classifiers](http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.3039) and [Transforming Classifier Scores into Accurate Multiclass Probability Estimates](http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.13.7457). The model finds a stepwise constant function (using the Pool Adjacent Violators Algorithm aka PAV) that minimizes the squared error.

#### Naïve Calibration

Naïve calibrator divides the range of the outputs into equally sized bins. In each bin, the probability of belonging to class 1 is the number of class 1 instances in the bin, divided by the total number of instances in the bin.

### Multi-class classification techniques

While some classification algorithms naturally permit the use of more than two classes, others are naturally [binary](http://en.wikipedia.org/wiki/Binary_classification) algorithms. However, binary classification algorithms can be turned into multi-class classification algorithms by a variety of strategies. TLC supports the two strategies described below for converting binary classification algorithms into multi-class classification algorithms.

#### One-vs-all

In this strategy, a binary classification algorithm is used to train one classifier for each class, which distinguishes that class from all other classes. Prediction is then performed by running these binary classifiers, and choosing the prediction with the highest confidence score.

#### Pairwise coupling (or all-vs-all)

In this strategy, a binary classification algorithm is used to train one classifier for each pair of classes. Prediction is then performed by running these binary classifiers, and computing a score for each class by counting how many of the binary classifiers predicted it. The prediction is the class with the highest score.

### Measuring the quality of the classifier

TLC supports the metrics described below for measuring the quality of a classifier.

#### Accuracy

The *accuracy* of a classifier is the proportion of correct predictions in the test set.

#### Log-loss

 The *log-loss* metric, is computed as follows:

image002.png

where image004.png is the number of instances in the test set and image006.png is the probability returned by the classifier if the instance belongs to class 1, and 1 minus the probability returned by the classifier if the instance belongs to class 0. If the classifier is a multi-class classifier, then image006.png is the probability returned by the classifier for the class the instance belongs to.

#### Log-loss reduction

The *log-loss reduction* (also known as *relative log-loss*, or *reduction in information gain - RIG*) metric, computes the log-loss on the test set, and then scales it relative to a classifier that predicts the prior for every example:

image008.png

This metric can be interpreted as the advantage of the classifier over a random prediction. E.g., if the RIG equals 20, it can be interpreted as “the probability of a correct prediction is 20% better than random guessing”.

#### Positive/negative precision/recall

For binary classifiers, TLC also outputs positive and negative *precision and recall*. For the purpose of these definitions, assume that class 1 instances are also referred to as positive, and class 0 instances are referred to as negative.

Positive precision is the proportion of *correctly predicted positive* instances among all the *positive predictions* (i.e., the number of positive instances predicted as positive, divided by the total number of instances predicted as positive). Positive recall is the proportion of *correctly predicted positive* instances among all the *positive* instances (i.e., the number of positive instances predicted as positive, divided by the total number of positive instances). Negative precision and recall are defined similarly.

TLC can automatically generate [precision/recall curves](#choosing_the_output_options_htm__514). There are two curves generated: The *ROC curve* and the *PR curve*. The ROC curve plots the positive recall vs. the false positive fraction (i.e., the number of negative instances predicted as positive divided by the total number of negative instances. This is also equal to one minus the negative recall). The positive recall and the false positive fraction are computed at different thresholds (the threshold is the value above which an instance is considered positive and below which an instance is considered negative). The PR curve plots the positive precision vs. the positive recall, at various thresholds. TLC also generates a text file containing the positive recall, positive precision and false positive fraction for every threshold that was used in the plots. If [cross validation](#learning_modes_cross_validation__4676) is used, these values are averaged over the different classifiers that were trained by the different *folds*. The standard deviation of these three values is also reported (if training was done using a training dataset and a separate test dataset, these are 0), as well as the average over the folds of the classifier output that was used as the threshold.

#### Area under curve (AUC)

*Area under the curve* (or *AUC*) is another metric computed by TLC for binary classifiers. It is the area under the [ROC curve](#introduction_classification_meas_1343). The area under the ROC curve is equal to the probability that the classifier ranks a randomly chosen positive instance higher than a randomly chosen negative one (assuming 'positive' ranks higher than 'negative').

#### Micro-average and macro-average

In multi-class classification, TLC outputs two forms of accuracy. *Macro-average* is computed by taking the average over all the classes of the fraction of correct predictions in this class (the number of correctly predicted instances in the class, divided by the total number of instances in the class). *Micro-average* is the fraction of instances predicted correctly. The distinction between these two measures is useful, for example, if some of the classes are under-sampled: The Macro-average metric gives the same weight to each class, no matter how many instances from that class the dataset contains. The micro-average metric weighs each class according to the number of instances that belong to it in the dataset.

## Regression

Regression algorithms are algorithms that learn to predict the value of a real function on an instance of data. Their input is a set of labeled examples. Each example is represented by a feature vector, and each label is a real number. A regression algorithm trains a regressor using the training examples, which can then be used to predict the value of the function on new unlabeled instances.

### Measuring the quality of the regressor

There are four metrics provided by TLC to measure the quality of the regressor in a regression problem.

#### The L1 loss (or absolute loss)

This loss is defined as

image002.png

where image004.png is the number of instances in the test set, the image006.pngs are the predicted labels for each instance, and the image008.pngs are the correct labels of each instance.

#### The L2 loss (or squared loss)

The L2 loss is defined as

image010.png

#### Root mean square loss (or RMC)

The *root mean square* loss (or *RMC*) is the square root of the L2 loss.

#### User defined loss function

This is the average of a loss function defined by the user, computed over all the instances in the test set.

## Quantile Regression

Quantile regression is a type of regression analysis. Whereas regression results in estimates that approximate the conditional *mean* of the response variable given certain values of the predictor variables, quantile regression aims at estimating either the conditional median or other quantiles of the response variable.

When using the quantile regressor tester, the quantiles can be viewed in the [per instance](#choosing_the_output_options_htm__5399) output file. By default, these are the five quantiles reported by TLC for each instance:

* the sample minimum (smallest observation)
* the lower quartile or *first quartile*
* the median (middle value)
* the upper quartile or *third quartile*
* the sample maximum (largest observation)

The overall results give the same metrics as the regression testers.

## Anomaly Detection

Anomaly detection (or **outlier detection**) is the identification of items, events or observations which do not conform to an expected pattern or other items in the dataset. Typical examples of anomaly detection tasks are detecting credit card fraud, medical problems or errors in text. Anomalies are also referred to as outliers,  novelties, noise, deviations and exceptions.

There are four metrics provided by TLC to measure the quality of anomaly detectors. The first three are detection rates. The detection rate at D is computed by sorting the test examples by the output of the anomaly detector in descending order, and then computing the number of positive examples out of the first D examples, divided by the total number of positive examples.

### Detection rate at k false positives

When the test examples are sorted by the output of the anomaly detector in descending order, denote by K the index of the k'th example whose label is 0. Detection rate at k false positives is the detection rate at K.

### Detection rate at fraction p false positives

When the test examples are sorted by the output of the anomaly detector in descending order, denote by K the index such that a fraction p of the label 0 examples are above K. Detection rate at fraction p false positives is the detection rate at K.

### Detection rate at number of anomalies

Denote by D the number of label 1 examples in the test set. Detection rate at number of anomalies is equal to the detection rate at D.

### AUC

This metric is described [here](#introduction_classification_meas_1203).

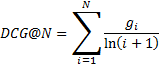
## Ranking

Ranking is a problem in which the goal is to automatically construct a ranker from a set of labeled examples. This set consists of [groups of instances](#input_data_format_ranking_instan_419), with some specified between instances in each group. This order is typically induced by giving a numerical or ordinal score or a judgment (e.g. degrees of relevance: "perfect", “good”, “fair”, “bad”) for each instance. The purpose of ranking algorithms is to train a ranker that can rank new groups of instances for which the score of each instance is unknown.

### Measuring the quality of the ranker

In ranking data sets, the label of each instance is either an integer between 0 and 4, or one of the relevance labels “bad”, “fair”, “good”, “excellent” or “perfect”. Each one of these labels is associated with a numerical gain. The gain can be [specified by the user](#tlc_learners_fasttree_htm_label__2299), but its default values are 0 (for “bad” or 0), 3 (for “fair” or 1), 7 (for “good” or 2), 15 (for “excellent” or 3) and 31 (for “perfect” or 4).

To measure the quality of the ranker, there are two available metrics; *DCG@N* (*Discounted Cumulative gain*) and *NDCG@N* (*Normalized Discounted Cumulative Gain*), where N is an additional integer parameter. DCG is computed as follows:



where image004.png is the gain associated with the relevance label of the instance ranked in the image006.png’th position. Note that TLC uses image008.png in contrast to image010.png(Wikipedia version). This change impacts DCG values.

NDCG is defined as:  
image012.png

where image014.png is the value of DCG@N when the documents are ordered in the ideal order from most relevant to least relevant. In case there are ties in scores, metrics are computed in a pessimistic fashion. In other words, if two or more results get the same score, for the purpose of computing DCG and NDCG they are ordered from least relevant to most relevant.

**Note:** if the ranked group of instances has image016.png instances and image018.png, then image020.png and image022.png are computed in the same way as image024.png and image026.png respectively.

# Getting started

TLC package is available for download as a zip file at <http://tlc/download>. Post download extract files from the zip archive to a local folder.

Get the TLC quick start guide [here](http://tlc/doc/QuickStart.pdf).

See some simple command lines [here](#running_experiments_running_expe_175).

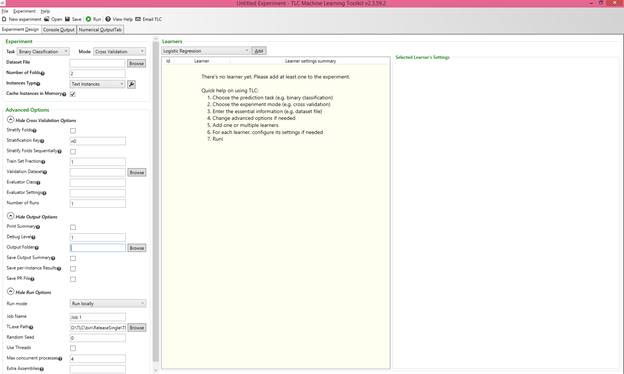
Join the quarterly TLC News by [*clicking here*](http://idwebelements/GroupManagement.aspx?Group=tlcnews&Operation=join).

Join the TLC Discussion DL by [*clicking here*](http://idwebelements/GroupManagement.aspx?Group=tlcdisc&Operation=join)*.*

To contact just the TLC team, email [tlcsupp](mailto:tlcsupp).

As mentioned in the introduction, the package includes TLC.exe (a GUI variant) and TL.exe (a command line variant).

Snapshot of TLC GUI:



Hovering with the mouse over a question mark icon displays a tooltip for the field next to it. The options in the Experiment box in the left pane of the GUI (task, learning mode, the number of folds in cross validation, train and test datasets) are saved in the registry, so that they persist when TLC is closed and reopened. The name of the output folder and the run options (the name of the HPC cluster, thread usage, the maximal number of concurrent processes, the random seed and extra assemblies) are saved in the registry as well.

Executing TL.exe ? on the command line displays a complete list of options and their default values.

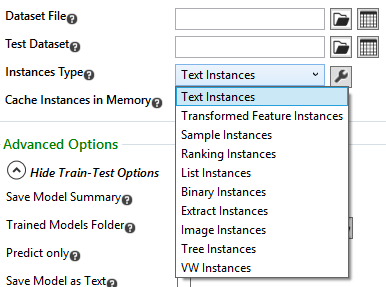
TLC is composed of configurable components; data parsers, data writers, data normalizers, learners, evaluators (which compute the different metrics of models on test data) etc. For each of these components, executing TL.exe ? ComponentClassName displays a list of options available for this component and their default values. For example:

TL.exe ? BNN (for the options of the Binary Neural Networks algorithm)  
TL.exe ? MinMaxNormalizer (for options of the min-max normalizer)  
TL.exe ? BinaryInstanceWriter (for options of the binary instance writer)  
TL.exe ? ClassifierTester (for options of the evaluator for binary classifiers)

When TLC is run using the GUI, it prints the equivalent command-line in the results tab. The command-line can be examined or copy-pasted from there. When running TLC from the command line, the parameters are order-insensitive. Parameters can also be read from a *settings file* instead (also known as a *response file*), by specifying @<SettingsFile>. See a sample settings file here: [\\cloudmltlc\TLC\Samples\MNIST\convolve.rsp](file:////cloudmltlc/tlc/samples/mnist/convolve.rsp). This scheme is the same scheme used by compilers, e.g, the C# compiler: csc.exe @MyBuild.rsp.  Note that even when using a response file, additional settings can be specified on the command line, e.g TL.exe @convolve.rsp /cl NN {accel=gpu}.

# Input data format

TLC can read input files in different formats. This section discusses the most common formats. Each format has its own set of user defined parameters that can be specified by clicking the image001.png button on the right of the Instances Type drop list:



*On the Command Line:*

* Use the command-line argument /inst followed by the Instances Type class name to pass the data format to the program
* Specifying non-standard format is done using an *instances parsing settings string* (referred to throughout the manual as the *IPS string*)
* the IPS string is a string in the format parameter1=value1 parameter2=value2 ...
* the IPS string is passed to the program after the Instances Type class name in a pair of curly brackets

In addition to the input format, feature transformations can also be specified by using Transformed Feature Instances.

**Tip:**  It is recommended to convert data once to [TLCBIN format](#input_data_format_binary_input_f_7440) and then run the experiments on the file containing the data in this format. This can be done on the command line using [CreateInstances](#other_tlc_modes_createinstances__1050) mode.

## Binary input format (TLCBIN)

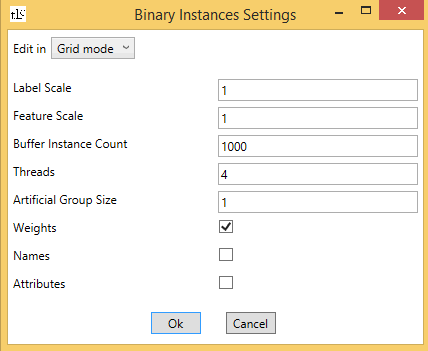
TLC provides a compact format to represent datasets. Any dataset can be converted into this format using [CreateInstances](#other_tlc_modes_createinstances__1050) mode.

To train or test using a dataset in TLCBIN format, Binary Instances must be specified in the [Instances Type](#learning_modes_train_mode_htm_in_1141) drop list in the experiment box in the left pane of the TLC GUI.

*On the Command Line:*

Use the command-line argument /inst BinaryInstances to specify TLCBIN input format.

This is the dialog box for the settings of Binary Instances:



### Label Scale

The label of every instance streamed is multiplied by this value. Default value is 1.

*On the Command Line:*

Add lscale= followed by its value to the IPS string to change the value of the label scale.

### Feature Scale

All features of every instance streamed are multiplied by this value. Default value is 1.

*On the Command Line:*

Add fscale= followed by its value to the IPS string to change the value of the feature scale.

### Buffer Instance Count

This specifies the maximum number of instances to stream into memory at any given point in time. Default value is 1000.

*On the Command Line:*

Add bic= followed by its value to the IPS string to indicate the number of instances to stream into memory at a given point in time.

### Threads

This specifies the number of worker decompressor threads to use. Default value is 4.

*On the Command Line:*

Add t= followed by the number to the IPS string to indicate the number threads to use for decompression.

### Artificial group size

If the binary instances were created from [Tree Instances](#input_data_format_bing_osd_bin_f_4931), or from [Text Instances](#input_data_format_text_input_for_6144) using a [grouping key](#input_data_format_ranking_instan_419), the data is composed into groups by the query or the grouping key (respectively). In all other cases, the data is artificially composed into groups of size specified in this text box. The default value is 1.

*On the Command Line:*

Add gs= followed by its value to the IPS string.

### Weights

This flag indicates whether the weight of each instance should be read. By default the check box is checked.

*On the Command Line:*

Add w=- to the IPS string to indicate that instances do not have weights.

### Names

This flag indicates whether the name of each instance should be read. By default the check box is unchecked.

*On the Command Line:*

Add names=+ to the IPS string to indicate that the instance names should be read.

### Attributes

This flag indicates whether the attributes of each instance should be read. By default the check box is unchecked.

*On the Command Line:*

Add attr=+ to the IPS string to indicate that the instance attributes should be read.

## Text input format

The default input format of TLC is text format. To use this format, choose Text Instances in the [Instances Type](#learning_modes_train_mode_htm_in_1141) drop list in the experiment box in the left pane of the TLC GUI.

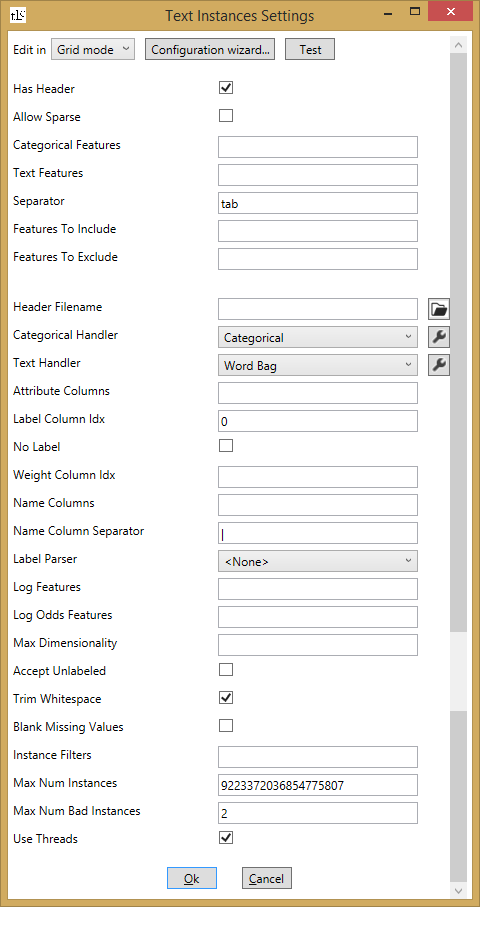
*On the Command Line:*

Use the command-line argument /inst TextInstances to specify text format.

TLC defines a standard input file format. The standard format is one instance per line, each instance consisting of tab separated feature values.

* An optional instance name (a string preceded by ‘\_’).
* By default, the first column (or the second if the first column contains a string beginning with ‘\_’) contains the label of the instance.
  + 0/1 for binary classification.
  + An integer between 0 and image003.png for multiclass classification with image005.png classes.
  + A real number for regression.
  + An integer between 0 and 4, or one of the strings “Bad”, “Fair”, “Good”, “Excellent”, “Perfect” for ranking.
* The rest of the features, tab-separated values, or tab-separated pairs index:value if using [sparse format](#input_data_format_text_input_for_6037).

Next, the settings of Text Instances are discussed:



The settings can be defined in the above dialog box, or alternatively, they can be passed by choosing “Text mode” from the drop list on the top of the dialog box and writing the IPS string the text box.

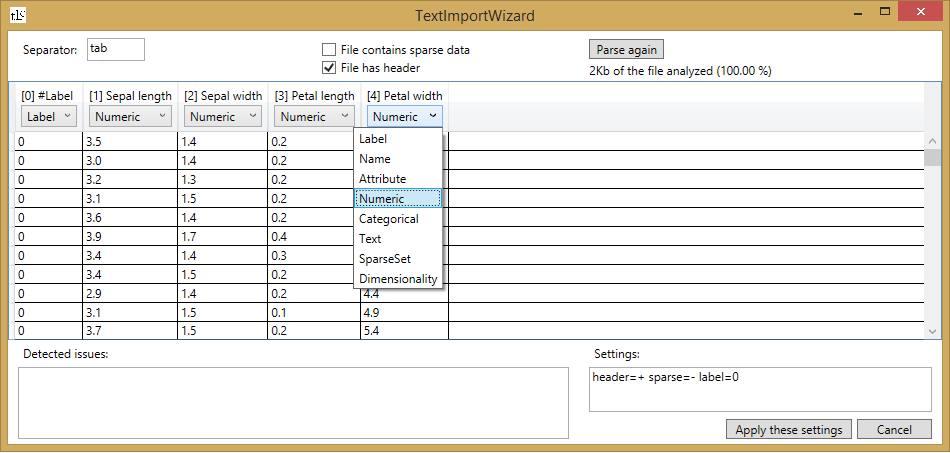
In order to help find the correct settings for a given dataset, next to every dataset text box in the TLC GUI there is the button big.gif. Clicking this button launches Big.exe. Big.exe is a lightweight application to quickly browse any structured data (csv, tsv etc).

*On the Command Line:*

Big.exe can be launched from the command-line, either by writing Big.exe, and then specifying the file in the window that opens, or by writing Big.exe followed by the file name. For example:

Big.exe  [\\cloudmltlc\TLC\Samples\UCI\adult.train](file://///cloudmltlc/TLC/Samples/UCI/adult.train)

Another way to discover the correct settings for a given dataset, is to use the Configuration wizard button in the Text Instances Settings dialog box. Click this button to load the first 4 MB of the file. The wizard will attempt to auto-detect the parameters and gives you a visual interface for editing them:



You can override the settings and it will automatically be reflected in the auto-generated settings string. The "Detected issues" box shows hints and warnings about the current setup.

### Has Header

The first line in the file may be a header line, which specifies the names of the columns (these names are used when, for example, an algorithm outputs its weight for each feature). If the input file has a header line, check the Has Header check box.

*On the Command Line:*

Add header=+ to the IPS string (or, in short form: h+).

### Allow Sparse

The instance parser auto detects [sparse instances](#input_data_format_text_input_for_6037) by detecting strings in the format index:value. If the dataset contains this format not as part of a sparse instance representation (for example, as part of a text feature), then the Allow Sparse check box must be unchecked to prevent a parsing error.

If the dataset contains index:value substrings both as part of text features and in sparse instance representations, the former must be coded by a different string and the Allow Sparse check box must remain checked to prevent a parsing error.

*On the Command Line:*

Add sparse=- to the IPS string to indicate dataset contains no sparse instances, and may contain text in the format integer:string.

### Categorical Features, Text Features, Categorical Handler, Text Handler

TLC can parse textual features in two ways: either interpreting them as categorical features or as text features. For each of these methods, TLC provides two handlers, described below.

#### Categorical features

The columns to be interpreted as categorical features are specified in a comma separated list of indices in the Categorical Features text box. The index list may contain ranges of indices. For example: 1,3,5-9,12. The columns may alternatively be defined in the settings of the handler.

*On the Command Line:*

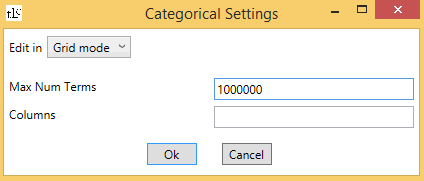
Add cat= followed by the list of indices to the IPS string.

#### Categorical handler

##### Categorical (categorical to binary features)

When features are parsed with this handler, an additional pass through the data is performed, to observe all the values of each categorical feature. Each categorical feature is replaced by a set of binary features representing all its possible values. For every instance, exactly one of the binary features in each set has the value 1 (the one corresponding to the instance’s value of the categorical feature) and the rest have the value 0. The name of each new binary feature is the name of the categorical feature, followed by ‘\_’ and the value of the feature.

The categorical feature handler has two user defined parameters. The first  indicates the maximal number of binary features to create. If there are more categories, instances that have values not represented by any binary feature, have the value 0 in all of them. The second indicates the set of columns to handle using this handler.



Alternatively, the parameter values can be passed using a *handler settings string*, by choosing “Text mode” from the drop list on the top of the dialog box  and writing the handler settings string in the format parameter1=value1 parameter2=value2 ….

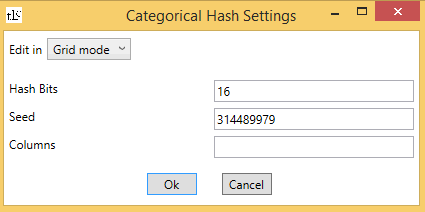
*On the Command Line:*

* Add ch=Categorical followed by the handler settings string in a pair of curly brackets
* To change the maximal number of categories add max= followed by its value to the handler settings string
* To specify the columns to handle, add cols= followed by the list of indices to the handler settings string. If no indices are specified, the indices specified by cat= are handled
* To add more handlers with different parameter values, add handler=Categorical followed by the handler settings string in a pair of curly brackets to the IPS string

##### Categorical Hash

When features are parsed with this handler, the categorical features associated with the handler are replaced by a set of numeric features of size 2^(the number of hash bits). For every instance, the value of the i'th feature created is computed as the number of columns that contain values that map to i when the hash function is applied to them. Note that the original feature names are not saved when using categorical feature hashing.

The categorical hash feature handler has three user defined parameters. The first  indicates the number of bits to map the categorical values to. The second indicates the seed for the hash function and the third indicates which columns to handle using this handler.



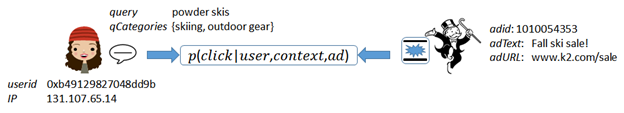
Alternatively, the parameter values can be passed using a *handler settings string*, by choosing “Text mode” from the drop list on the top of the dialog box  and writing the handler settings string in the format parameter1=value1 parameter2=value2 ….

*On the Command Line:*

* Add ch=CategoricalHash followed by the handler settings string in a pair of curly brackets
* To specify the number of hash bits, add bits= followed by the number of bits to the handler settings string
* To specify the seed of the hash function, add seed= followed by the seed to the handler settings string
* To specify the columns to handle, add cols= followed by the list of indices to the handler settings string. If no indices are specified, the indices specified by cat= are handled
* To add more handlers that map to a separate set of hash features, add handler=CategoricalHash followed by the handler settings string in a pair of curly brackets to the IPS string

##### Count handler (Dracula – learning with counts)

The count handler is a handler that can only be specified from the command line.

Scenario : Large-Scale learning with high-cardinatlity attributes or categorical features. 

In these cases feature space can explode to 100’s of millions and learning is hard. Representing the attribute values with the class-conditional counts and learning with those counts is a neat way to avoid the explosion in feature space. This approach has many practical advantages - its scalable, efficient and adaptive ! TLC makes it super-easy for users to try out this powerful technique on their problems via a feature handler called  Dracula.

Currently the handler assumes two classes, so it only works for binary classification problems. These are the settings for the count handler:

###### Label Column

This is a required field. It is the column based on which the class-conditional counts are calculated.

###### binMapper

The binmapper maps the value in the column to the corresponding key/bin that the value should fall into. The options are

* Hash Based mapper - Simply hashes the raw value of the column to an integer.
* IntBin - applicable to float valued columns. Maps the float values into equi-density bins. Uses the same algorithm as is used by the [Bin Normalizer](#ml_basics_feature_normalization__9206).

###### CountTable

This is the subcomponent that does the counting and finally produces the features. It uses the label information and counts class-specific counts for each mapped column value (by the bin mapper). It then converts the counts into features.

* Array based count table. Stores the + and - counts for each possible value of the mapped feature. This count table has three parameters:
  + smoothingCount: the default number of count-per-class assigned when calculating log-odds. Min value is 1, default value is 1.
  + laplaceScale: Laplacian noise diversity/scale-parameter. Suggest keeping it less than 1. The default value is 0.1.
  + seed: the seed for the random number generator for the Laplacian noise. The default value is 314489979.
  + garbageThreshold: if total number of examples with the given feature value is below the threshold, these examples are all grouped together to form a ‘garbage bin’. These ‘garbage counts’ are then assigned to all examples that have total number of occurrences below the threshold. Default value is 0, which means that garbage bins are disabled.
* CountMinSketch based table. Creates a countminsketch for the mapped feature values. Scales better when the number of possible mapped feature values is very high. This count table has five parameters:
  + depth: count-min sketch table depth. Default value 4.
  + width: count-min sketch width. Default value 8388608.
  + smoothingCount: default number of count-per-class assigned when calculating log-odds. Min value is 1, default value is 1.
  + laplaceScale: Laplacian noise diversity/scale-parameter. Suggest keeping it less than 1. Default value is 0.1.
  + seed: seed for the random number generator for the Laplacian noise.
  + garbageThreshold: if total number of examples with the given feature value is below the threshold, these examples are all grouped together to form a ‘garbage bin’. These ‘garbage counts’ are then assigned to all examples that have total number of occurrences below the threshold. Default value is 0, which means that garbage bins are disabled.
* Dictionary based table. Stores the + and - counts for the seen features values in a dictionary. This count table has three parameters:
  + smoothingCount: default number of count-per-class assigned when calculating log-odds. Min value is 1, default value is 1.
  + laplaceScale: Laplacian noise diversity/scale-parameter. Suggest keeping it less than 1. Default value is 0.1.
  + seed: seed for the random number generator for the Laplacian noise.
  + garbageThreshold: if total number of examples with the given feature value is below the threshold, these examples are all grouped together to form a ‘garbage bin’. These ‘garbage counts’ are then assigned to all examples that have total number of occurrences below the threshold. Default value is 0, which means that garbage bins are disabled.

         laplaceScale is a parameter on all the CountTables. A value above 0 indicates that some noise will be added to the counts when doing the actual training. This to prevent overfitting when counts are calculates on the same dataset as the one that is used for learning based on the count-based features.

*On the Command Line:*

* Add handler=Dracula followed by the handler settings string in a pair of curly brackets to the IPS string
* Add label= followed by the label columns index to the handler settings string
* Add binMapper= followed by HashMapper or IntBin to the handler settings string. The bin mapper settings string is passed after its name in a pair of curly brackets. The hash mapper has one parameter: hashbits, indicating the number of bits to hash into. The int bin mapper has one parameter: maxBins, indicating the maximum number of bins to map the real-valued column to
* Add table= followed by Array, CMTable or DictTable to the handler settings string. The count table settings string is passed after its name in a pair of curly brackets. The common parameters to all tables are smoothingCount, laplaceScale, seed and garbageThreshold. the CountMinSketch table has two additional parameters: depth and width.
* Example:  
  TL.exe  \\cloudmltlc\TLC\Samples\UCI\adult.train /inst Text { header+ sep=, label=14 handler=Dracula{cols=5-9 label=14 binMapper=hash table=Dict{laplace=.2 smooth=3}} } /cl LR

#### Text features

Some data may contain textual features (e.g., in email classification, instances may contain a subject feature and a body feature. Another example is when a field contains arbitrary features produced by some external feature producer). Specifying the text features is done in a comma separated list of indices in the Text Features text box. The index list may contain ranges of indices. For example: 1,3,5-9,12. The columns may alternatively be defined in the settings of the handler.

*On the Command Line:*

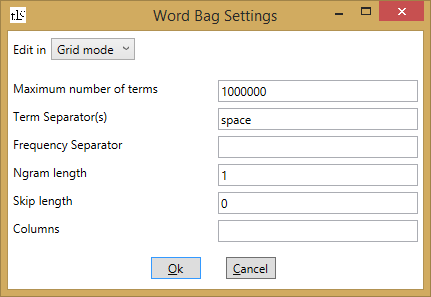
Add text= followed by the list of indices to the IPS string.

#### Text handler

##### Word bag

When features are handled using this handler, an additional pass through the data is performed, where for every text feature, a set of numeric features corresponding to the separate words in the text is created. This method of handling text features is known as “bag of words”. The value each numeric feature is the number of occurrences of the word in the text. The names of the numeric features are the name of the text feature, followed by ‘\_’ and the substring. The text can also contain substrings of the form “word:count”, where count is an integer. This adds count to the number of occurrences of word in the text. **Note**: If the text contains substrings of the form “word:count”, the colon must be specified explicitly in the "Frequency Separator" text box (see below), otherwise these substrings are parsed as a word in the word bag. A more general way of handling text features is known as N-grams, where the numeric features generated are the number of occurrence of sequences of words of length N. TLC supports 1-grams, which is the simple bag of words; n-grams, which counts occurrences of n consecutive words; and skip-grams, which is similar to n-grams but the words need not be in consecutive order.

The text feature handler has four user defined parameters. Term Separator(s) indicates the separator or separators used to separate the text into words (the default separator is space). If more than one separator is specified, they are comma separated. Frequency Separator indicates the separator used to separate between a word and its count (the default separator is ‘:’). Max Num Terms indicates the maximal number of features that are created by the parser, and Columns indicates which columns to handle using this handler.



Alternatively, the parameter values can be passed using a *handler settings string*, by choosing “Text mode” from the drop list on the top of the dialog box  and writing the handler settings string in the format parameter1=value1 parameter2=value2 ….

For example, suppose you have a dataset containing two text features named F1 and F2, and the following two instances:

|  |  |  |
| --- | --- | --- |
| Instance | F1 | F2 |
| 1 | foo bar | foo alpha:1 beta:5 |
| 2 | bar foobar bar | beta:2 beta |

The numeric features that are created for this dataset are F1\_foo, F1\_bar, F1\_foobar, F2\_foo, F2\_alpha and F2\_beta. The values are as follows:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Instance | F1\_foo | F1\_bar | F1\_foobar | F2\_foo | F2\_alpha | F2\_beta |
| 1 | 1 | 1 | 0 | 1 | 1 | 5 |
| 2 | 0 | 2 | 1 | 0 | 0 | 3 |

*On the Command Line:*

* Add th=WordBag followed by the handler settings string in a pair of curly brackets
* To specify the term separators add csep= followed by the separator to the handler settings string
* For more than one separator, specify them separated by commas
* To specify the frequency separator add sep= followed by the separator to the handler settings string
* To specify n-grams add ngram=  followed by the maximum number of words in the n-gram to the handler settings string
* To specify skip-grams add skips=  followed by the maximum number words in the skip-gram to the handler settings string
* To specify the maximal number of features to create add max= followed by the number to the handler settings string
* To specify the columns to handle, add cols= followed by the list of indices to the handler settings string. If no indices are specified, the indices specified by text= are handled
* To add more handlers with different parameter values, add handler=WordBag followed by the handler settings string in a pair of curly brackets to the IPS string

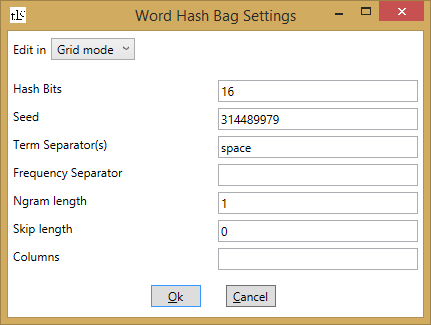
##### Word hash bag

For every instance, exactly one of the binary features in each set has the value 1 (the one corresponding to the value of the hash function applied to the instance’s value) and the rest have the value 0. Note that the original feature names are not saved when using categorical feature hashing.

When features are handled using this handler, the text features associated with the handler are replaced by a set of numeric features of size 2^(the number of hash bits). For every instance, the value of the i'th feature created is computed as the total number of occurrences, in the text in all the columns, of words that map to i when the hash function is applied to them. Note that the original feature names are not saved when using text feature hashing. The text can also contain substrings of the form “word:count”, where count is an integer. This adds count to the number of occurrences of word in the text.

Word hash bag also supports N-grams, where the numeric features generated are the number of occurrence of sequences of words of length N. TLC supports 1-grams, which is the simple bag of words; n-grams, which counts occurrences of n consecutive words; and skip-grams, which is similar to n-grams but the words need not be in consecutive order.

The text feature hashing handler has five user defined parameters. Term Separator(s) indicates the separator or separators used to separate the text into words (the default separator is space). If more than one separator is specified, they are comma separated. Frequency Separator indicates the separator used to separate between a word and its count (the default separator is ‘:’). Hash Bits indicates the number of bits to map to. Seed indicates the seed for the hash function and Columns indicates which columns to handle using this handler.



Alternatively, the parameter values can be passed using a *handler settings string*, by choosing “Text mode” from the drop list on the top of the dialog box  and writing the handler settings string in the format parameter1=value1 parameter2=value2 ….

*On the Command Line:*

* Add th=WordHashBag followed by the handler settings string in a pair of curly brackets
* To specify the term separators add csep= followed by the separator to the handler settings string
* For more than one separator, specify them separated by commas
* To specify the frequency separator add sep= followed by the separator to the handler settings string
* To specify n-grams add ngram=  followed by the maximum number of words in the n-gram to the handler settings string
* To specify skip-grams add skips=  followed by the maximum number words in the skip-gram to the handler settings string
* To specify the number of hash bits, add bits= followed by the number of bits to the handler settings string
* To specify the seed of the hash function, add seed= followed by the seed to the handler settings string
* To specify the columns to handle, add cols= followed by the list of indices to the handler settings string. If no indices are specified, the indices specified by text= are handled
* To add more handlers that map to a separate set of hash features, add handler=WordBag followed by the handler settings string in a pair of curly brackets to the IPS string

##### char-gram

Char-gram, or character n-grams is a simple text handler where each text is considered a bag of character n-grams, that is, strings of length n. An important characteristic of the character level n-grams is that they avoid the problem of sparse data that arises when using word level n-grams. That is, there is much less character combinations than word combinations, therefore, less n-grams have zero frequency. This handler is only available from the command line.

*On the Command Line:*

* Add handler=CharGram followed by the handler settings string in a pair of curly brackets to the IPS string
* Add len= followed by the n-gram length (2,3 or 4) to the handler settings string. The default value is 3
* Add lower=- or lower=+ to the handler settings string to indicate whether or not to map characters to lower case. The default is true
* Add max= followed by a number to the handler settings string to specify the maximum number of character n-grams to keep. The default value is 1000000
* Example: TL.exe /c TrainTest /cl LR LM\Local.source\_features.de-de.txt /test LM\Local.validate\_features.de-de.txt /inst Text{lf=LM\Mapping.de-de.txt header+ attr=1,2 handler=CharGram{cols=3,4 len=4 lower=-}}

### Separator

In the standard input format features are tab-separated. If a different separator is used, specify it in the Separator text box. If the separator is space, comma, colon, semicolon or tab, it can be passed to the parser using the name of the separator instead of the separator itself (for example: semicolon instead of ;).

*On the Command Line:*

Add sep= followed by the separator to the IPS string. Space, comma, colon, semicolon or tab can be specified by sep=space, sep=comma, sep=colon, sep=semicolon or sep=tab respectively.

### Feature exclusion and inclusion

It is possible to specify a subset of features to ignore, or, alternatively, a subset of features to use. This is done by specifying the features that should be excluded or included in the Features to Include or Features to Exclude text boxes. Features can be specified by index, name, substrings or regular expressions.

* To specify by index, use i: followed by a list of comma-separated indices. The list can include ranges of indices  
  **Note:** the indices are 0-based, and only count the **feature** columns, and not the name/attribute/label columns. E.g, if the header line of the dataset is: f0 label n0 f1 n1 n2 f2 f3 f4 (i.e, each instance has a feature in the first column, its label in the second, a name in the third, fifth and sixth columns and features in the rest), then to exclude the feature f1 specify i:1. Specifying i:3 would exclude the feature f3.
* To specify by name, use e: followed by a comma-separated list of feature names. In datasets that do not have a header line, the features are given the names f0, f1, f2 and so on. If a name of a feature that does not exist is specified a warning is displayed, and training continues by including or excluding the features that were found.
* To specify by substring, use s: followed by a comma-separated list of substrings. The features with names containing one of the substrings are included/excluded.
* To specify by regular expression, use r: followed by a comma-separated list of regular expressions. Features with names that match these are included/excluded.

The last two options are useful in cases where there are numerical features produced by the parser that correspond to categorical/text features, since the names given to these features all include the original feature name as a substring.

Excluding and including features can also be done by defining the features to be excluded as [attribute features](#input_data_format_text_input_for_9658).

*On the Command Line:*

Add incl= or excl= followed by i:, e:, s: or r: as explained above, and the list of indices, strings or regular expressions to the IPS string.

**Note:** The include/exclude features options currently cannot be used on files that have categorical or text features.

### Header Filename

The header containing feature names may be specified in a separate file. The file can contain comments beginning with "#" or "\\", and then the header in one line, separated using the same [separator](#input_data_format_text_input_for_8806) specified for the data file.

*On the Command Line:*

Add hf= followed by the header file name to the IPS string.

### Metadata: names and attributes

Often instances carry additional information alongside the features used for classification. TLC allows specifying two types of such metadata:  *names*, and *attributes***.** The difference is that columns specified as *names* are written out alongside the prediction in case the prediction results are outputted. Attributes are ignored.

#### Names, Name Column Separator

By default, the first column of each instance can be a name starting with ‘\_’. This is auto-parsed. It is possible to explicitly specify additional columns comprising the name. This is done by specifying a comma separated list of the indices of the columns comprising the name in the Name Columns text box.  The index list may contain ranges of indices. For example: 1,3,5-9,12.

If a single name column is specified (either by writing it in the first column preceded by ‘\_’, or by specifying it explicitly), its value is used as the instance’s ID in the output files. If multiple name columns are specified, the Name Column Separator option regulates what is seen in the output files. If it is set to none or null, the name columns are not concatenated, and appear in the output file in separate columns. If it is set to another string, a single name is formed by concatenating the name columns with the specified string as a separator. The default separator string is “|”.

Alternatively, metadata (names and attributes) can be stored in JSON format in the first column, with any embedded structure.

*On the Command Line:*

* Add name= followed by the list of indices of columns comprising the name to the IPS string
* Add ncsep= followed by the name column separator to the IPS string

#### Attributes

Specifying the attribute columns is done by specifying a comma separated list of the indices of the attribute features in the Attribute Columns text box. The index list may contain ranges of indices. For example: 1,3,5-9,12.

Alternatively, metadata (names and attributes) can be stored in JSON format in the first column, with any embedded structure.

*On the Command Line:*

Add attr= followed by the list of indices of attribute columns to the IPS string.

### Label column Idx

By default, the parser assumes that each instance’s label is either in the first column, or in the first numeric column if the first columns contain names. If the labels in the input file are in any other column, the label column must be specified in the Label Column Idx text box.

*On the Command Line:*

Add label= followed by the 0-based index of the label column to the IPS string.

### No Label

This check box is unchecked by default. If the dataset has no label column (for example, if it is an unlabeled dataset to make predictions on), check the No Label check box.

*On the Command Line:*

Add noLabel=+ to the IPS string to indicate that the dataset has no labels (the default is false).

### Weight column Idx

Some algorithms (such as FastRank) support learning with weighted examples. By default each example has a weight of 1. Setting an example’s weight to 3 is equivalent to duplicating it 3 times. To specify that examples are weighted, specify the 0-based index of the weight column in the Weight Column Idx text box.

*On the Command Line:*

Add weight= followed by the 0-based index of the weight column to the IPS string.

### Label Parser

The data input file may contain labels that are not numerical values (e.g., the strings “perfect”, “good”, “bad” in ranking problems). In this case, a mapping must be provided in order to convert each label to its corresponding numeric value. There are three ways of specifying the mapping:

* Provide a file containing the mapping. To use this option, choose File Label Parser from the Label Parser drop list. The mapping file contains one line for every label, each line in the format Label<tab>DoubleValue. For example, if the labels in the input file are “perfect”, “good” and “bad”, the mapping file would contain the following three lines:  
  perfect     3  
  good  2  
  bad   1
* Provide the mapping in a string. To use this option, choose String Label Parser from the Label Parser drop list. The mapping contains the different labels separated by commas, each label in the format Label=DoubleValue. For example, if the labels in the input file are "perfect", "good" and "bad", the mapping string would be perfect=3,good=2,bad=1.
* Map the labels automatically. To use this option, choose Automatic Label Parser from the Label Parser drop list. When this label parser is used, the first label seen in the dataset is mapped to 0, the second is mapped to 1, and so on. When this label parser is used, the label mapping is saved in text format in the LabelParser folder in the zip file created by the [/m option](#learning_modes_train_mode_htm_sa_1226). The format of the text file is identical to the format of the mapping file described above.

*On the Command Line:*

* To use File Label Parser, add lp=File { lf=FileName } (where FileName is the name of the file containing the mapping) to the IPS string
* To use String Label Parser, add lp=String { lm=Mapping } (where Mapping is the mapping string) to the IPS string
* To use Automatic Label Parser, add lp=Auto to the IPS string

### Log Features and Log Odds Features

TLC supports adding new features to the dataset. The new features that can be added are the natural logarithms of any existing features, and the difference between the natural logarithms of any pair of existing features. The features for which new features should be created are specified in a comma separated list of indices in the Log Features or the Log Odds Features text boxes.

*On the Command Line:*

* Add log= followed by a comma separated list of columns to the IPS string to indicate for which columns a new log feature should be created
* Add logodds= followed by a comma separated list of columns to the IPS string to indicate for which pairs of columns a new feature should be created

### Sparse and dense formats, Max Dimensionality

In many cases, some or all of the instances have many 0 valued features and only a few non-zero features. In this case, it is possible to represent the instance in a sparse format. In this format too, the columns are tab-separated.

* The first column, as in the dense format, contains an optional instance name.
* The next column is the instance label.
* The next column contains the dimension (i.e. the number of features including the 0 valued features) of the instance. Alternatively, specify the dimension in the Max Dimensionality text box. In this case the dimension column must be omitted.
* The next columns contain the non-zero features in the format index:value.

**Note:** the indices are 0-based.

**Note:** If the dataset contains text features (categorical, text, name or attribute features) their default value is an empty string.

**Note:** If all the features of a given instance are 0, there must be one column added after the dimension column, containing 0:0, otherwise this results in a parsing error.

*On the Command Line:*

Add maxdim= followed by the total number of features to the IPS string.

#### Example

Suppose a dataset contains three instances (x1,x2,x3), each having five features and a label (y1,y2,y3):

x1 = (0.4,0,0.1,0,0), y1 = 0  
x2 = (0,0.9,0,12,4577), y2 = 1  
x3 = (0,0,0,19,0), y3 = 1

The dense standard format representation of these instances is either (without the instance names):

0     0.4   0     0.1   0     0  
1     0     0.9   0     12    4577  
1     0     0     0     19    0

Or (with instance names):

\_x1   0     0.4   0     0.1   0     0  
\_x2   1     0     0.9   0     12    4577  
\_x3   1     0     0     0     19    0

The sparse format representation of these instances is either (without the instance names):

0     5     0:0.4 2:0.1  
1     5     1:0.9 3:12  4:4577  
1     5     3:19

Or (with instance names):

\_x1   0     5     0:0.4 2:0.1  
\_x2   1     5     1:0.9 3:12  4:4577  
\_x3   1     5     3:19

Specifying the maximum dimensionality (5) above must be omitted if the dimension is specified in the Max Dimensionality text box (or if maxdim=5 is added to the IPS string).

### Accept Unlabeled

By default, this check box is unchecked, indicating that unlabeled instances are treated as unparseable.

*On the Command Line:*

Add unl=+ to the IPS string to accept unlabeled instances.

### Trim whitespace

By default, this check box is checked, indicating that each line holding an instance has leading and trailing whitespace removed, before parsing begins.  Even if turned off linebreaks are never included in the parsing.

*On the Command Line:*

Add trim=- to the IPS string to suppress trimming whitespace.

### Blank Missing Values

By default, this check box is unchecked, indicating that feature fields with no content are treated as parse errors.  If checked, the feature is instead interpreted as a missing value.

*On the Command Line:*

Add blankmissing=+ to the IPS string to accept blank fields as missing values.

### Instance Filters

It is possible to define a subset of instances to be used for training by specifying *instance filters*: specific values for name or attribute features the instance must have in order to be used for training. An instance filter has the format ni=value, ni~value, ai=value or ai~value. n or a indicate name or attribute features. i indicates the 0-based index of the name or attribute feature as indicated in the [Name Columns text box or Attribute Columns text box](file://D:\TLC3\TLC_Main\doc\OnlineDocumentation\Input_data_format\Metadata:_names_and_attributes). = indicates instances must have this exact value and ~ indicates instances must contain this value.

For example, if the name columns are3,5,9,15 and attribute columns are17,23 then an instance filter can be n2~car to indicate training only with instances where the 9th column contains the string “car”, or a1=ML-1 to indicate training only with instances where the 23rd column is “ML-1”.

**Note:** To define more than one filter, ”Text mode” must be used (choose it from the drop list on the top of the Text Instances Settings dialog box). Add each filter separately by writing filter= followed by the filter in a pair of curly brackets. The filters are combined with an “or”. E.g, if there are two filters, n2~car and a1=ML-1, then instances with the 3rd name feature containing the string “car” **or** with the 2nd attribute feature equal to “ML-1” are included in the training process.

*On the Command Line:*

* Add filter= followed by the filter in a pair of curly brackets to the IPS string
* To add multiple filters, add each filter separately after filter= to the IPS string

### Max Num Instances

Indicates the maximal number of instances to be parsed.

*On the Command Line:*

Add max= followed by its value to the IPS string.

### Max Num Bad Instances

Indicates the maximal number of unparseable instances to skip before exiting and reporting an error.

*On the Command Line:*

Add maxbad= followed by its value to the IPS string.

### Use Threads

This check box indicates using threads to parse the data.

*On the Command Line:*

Add threads=- to the IPS string to indicate no threading or threads=+ to indicate threading.

### Example

The adult.train dataset contains data such as age, education, marital status, occupation, race, gender, etc., and this data is partitioned into two classes: people with salaries under $50,000 per year (this class label is 0), and people with salaries over $50,000 per year (this class label is 1). The file contains a header line, the feature values are separated by commas, and the labels are the last entry in each line, which is column number 14. The features in columns 1, 3, 5-9 and 13 are categorical. Therefore, the IPS string would be: header=+ label=14 cat=1,3,5-9,13 sep=,.

To use only *age* and *sex* features (in first and tenth columns), change the string to header=+ label=14 cat=1,3,5-9,13 sep=, attr=1-8,10-13.

This instructs the parser to ignore all the columns except for 0 and 9 (note that attr= overrides cat= features in the specification).

Suppose that the file adult.train.tlc contains the data in adult.train converted into [numeric format](#input_data_format_text_input_for_9140). This means that for each categorical feature in adult.train, there are numeric features beginning with the original feature’s name, followed by ‘\_’ and a value of this feature. In this case there are two equivalent ways to write the IPS string:

incl=e:age,sex\_Male,sex\_Female  
incl=i:0,2,64,65

Note that using the substring option incl=s:age,sex instead of the exact string option, would erroneously include the feature occupation\_Exec-managerial since it contains the substring ‘age’.

### Missing features

TLC supports a new handler called the 'MissingHandler' to handle columns  with potential missing or non-parseable values.

For every column it processes it replaces the original feature with two new features:

1. the actual parsed value (or 0  if it cannot parse the value)
2. A boolean feature set to 1 when the original feature is missing and 0 if it is not.

#### Examples

Tl.exe \\cloudmltlc\TLC\Samples\breast-cancer.txt /inst Text {handler=Missing{cols=3}}

Tl.exe \\cloudmltlc\TLC\Samples\breast-cancer.txt /inst Text {handler=Missing{cols=3,4-6 acceptany=-}}

If acceptAny is set to true(the default), the handler treats any non-parsable value as missing value. If it is set to false - only values of "?" or "NaN" are treated as missing and non-parsable values are treated as error.

## Ranking Instances

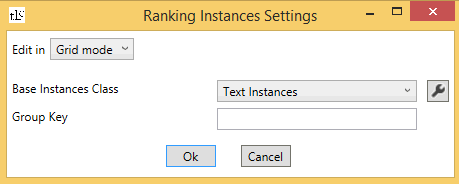
Ranking algorithms operate on groups of instances, rather than individual instances. Therefore, they require a feature that acts as a grouping key. For instance, in a web search application, each instance contains a query feature, a URL feature, a relevance score, etc. The query feature is the grouping key in this case (the goal is to rank web pages that are results of a single query).

To train or test a ranker, Ranking Instances must be specified in the [Instances Type](#learning_modes_train_mode_htm_in_1141) drop list in the experiment box in the left pane of the TLC GUI.

*On the Command Line:*

Use the command-line argument /inst RankingInstances.

This is the dialog box for the settings of Ranking Instances:



### Base Instances Class

Use this to specify the input format of the dataset.

*On the Command Line:*

* Add base= followed by the Instances class name to the IPS string to indicate the base Instances class
* Add the IPS string of the base Instances class following the class name in a pair of curly brackets

### Group Key

Specify the list of columns by which to group in the Group Key text box, separated by commas. The list can contain name features, indicated by n followed by the 0-based index of the name feature as indicated in the [Name Columns](#input_data_format_text_input_for_5383) text box, or attribute features, indicated by a followed by the index of the attribute feature as indicated in the [Attribute Columns](#input_data_format_text_input_for_9658) text box. For example, if the name columns are3,5,9,15 and attribute columns are 17,23 then if the Group Key text box contains the text n3,a1, the instances are grouped by the values in column 15 (which is the 4th column specified as a name feature) and column 23 (which is the 2nd column specified as an attribute feature).

*On the Command Line:*

Add groupKey= followed by the list of indices in the format described above to the IPS string.

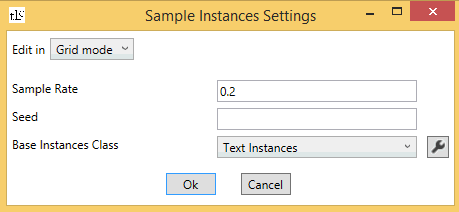
## Sample Instances

To train or test using a random sample of the dataset, specify Sample Instances in the [Instances Type](#learning_modes_train_mode_htm_in_1141) drop list in the experiment box in the left pane of the TLC GUI.

*On the Command Line:*

Use the command-line argument /inst SampleInstances.

This is the dialog box for the settings of Sample Instances:



### Sample Rate

Indicates the rate at which instances are sampled with.

*On the Command Line:*

Add rate= followed by its value to the IPS string.

### Seed

Indicates the seed for the random number generator used for sampling.

*On the Command Line:*

Add rs= followed by its value to the IPS string.

### Base Instances Class

Use this to specify the input format of the dataset.

*On the Command Line:*

* Add base= followed by the Instances class name to the IPS string to indicate the base Instances class
* Add the IPS string of the base Instances class following the class name in a pair of curly brackets

## Bing/OSD format

Many datasets in Bing (especially ones used in ranking/relevance training experiments) are currently saved in a binary format that is ideal for tree based algorithms – particularly FastRank. This format is called OsdBin/BingBin format. It is different than other TLC formats in that it is a column/feature based format as opposed to a row-based format. To use this format, specify Tree Instances in the [Instances Type](#learning_modes_train_mode_htm_in_1141) drop list in the experiment box in the left pane of the TLC GUI. The values of user defined parameters available for this format can be specified by clicking the image001.png button on the right of the Instances Type drop list.

See the [Bing wiki](http://bing/wiki/BinTools) for more details about this format.

*On the Command Line:*

* Use the command-line argument /inst OsdBin to specify this format
* Use the command-line argument /cacheinst- to disable caching all instances in memory
* Execute TL.exe ? OsdBin to display a list of parameters that can be defined for this format

## Bing Extraction format

Bing Extraction format is an input file format used for ranking problems. This format can be used by specifying Extract Instances in the [Instances Type](#learning_modes_train_mode_htm_in_1141) drop list in the experiment box in the left pane of the TLC GUI.

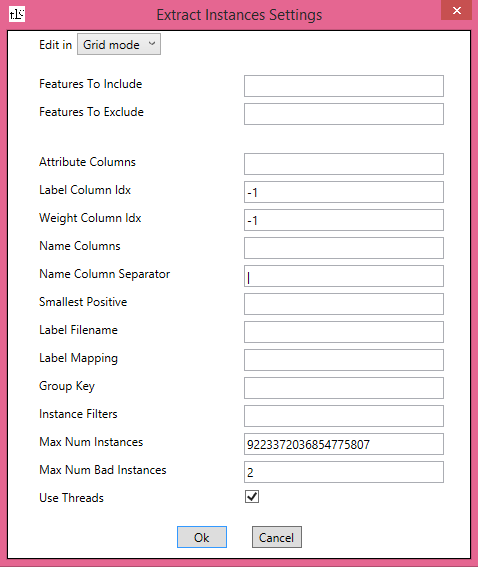
*On the Command Line:*

Use the command-line argument /inst ExtractInstances to specify Bing extraction format.

Data saved in this format contains a few columns with standard names that can automatically be parsed without the need to specify them in the IPS string:

* The column named “m:QueryId” is treated as the grouping key (if it does not exist, then the “m:Query” column is used).
* The label column is named “m:Rating”.
* If a column named “m:Weight” exists it is treated as the weight column.

This is the dialog box for the settings of Extract Instances:



This format has many settings in common with [text format](#input_data_format_text_input_for_6144) (excluding/including features, name and attribute columns, label and weight columns, label file name, grouping key, instance filters, maximal number of instances and bad instances and thread usage). In addition there are two more options.

### Label Mapping

The label mapping in Bing Extraction format can also be specified directly instead of in a file, in the Label Mapping text box. The format for specifying the mapping is label1=value1,label2=value2,….

If neither a label mapping file nor the label mapping are specified, the parser tries to find a file with the same name as the data file, where the suffix is replaced by “.txt.labels”. If this file exists, it is used as the label mapping file. If it does not exist, the default mapping is used:

Perfect=4, Excellent=3, Good=2, Fair=1, Bad=0, Detrimental=0, Unjudged=0, PageDidNotLoad=0, ERROR=0, 4=4, 3=3, 2=2, 1=1, 0=0, True=1, False=0, true=1, false=0, Definitive=4, NotRelevant=0.

*On the Command Line:*

Add lm= followed by the label mapping in the format label1=value1,label2=value2,... to the IPS string.

### Smallest Positive

When doing binary classification on datasets containing data in Bing Extraction format, it is possible to specify the minimal label (between 0 and 4) to be considered positive. By default, every label above ‘Bad’ (or 0) is considered positive. This threshold can be changed in the Smallest Positive text box.

*On the Command Line:*

Add pos= followed by an integer between 0 and 4 to the IPS string.

## Image input format

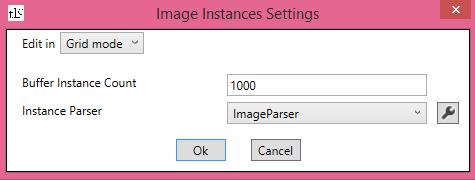
Many machine learning problems involve datasets where each instance is an image. TLC can be used on image datasets by specifying Image Instances in the [Instances Type](#learning_modes_train_mode_htm_in_1141) drop list in the experiment box in the left pane of the TLC GUI.

*On the Command Line:*

Use the command-line argument /inst ImageInstances to specify image format.

The dataset file format for training/testing on image files is a text format, with each line consisting of a full path to an image and its label, separated by a tab.

This is the dialog box for the settings of Image Instances:



### Buffer Instance Count

The Image Instances type supports parsing of images in the background while training on an already parsed image instance. The value of this parameter dictates the number of instances to hold in memory at any given point in time. By default, the number of image instances cached in memory is 1000.

**Tip:** Since the datasets created are huge, when using ImageInstances it is recommended to uncheck the [Cache Instances in Memory](#learning_modes_train_mode_htm_ca_2710) check box and use this parameter instead.

*On the Command Line:*

* Add bic= followed by an integer to the IPS string to indicate the number of instances to cache
* Use the command-line argument /cacheinst- to disable data caching (recommended)

### Image instance parser and settings

The ImageParser loads each image as a c# Bitmap object, scales it to the dimensions specified in the parser arguments and builds a feature vector representing the image pixels. These arguments can be specified by clicking the image004.png button next to the Instance Parser drop list (the drop list contains only one option). Alternatively, the Image Parser settings can be passed using a *parser settings string*, by choosing “Text mode” from the drop list on the top of the dialog box  and writing the parser settings string in the format parameter1=value1 parameter2=value2 ….

*On the Command Line:*

Add ip=ImageParser followed by the parser setting string in a pair of curly brackets to the IPS string.

These are the parameters that can be defined for the parser:

#### Width and Height

These parameters define the scaling of the image, with default values of 256.

*On the Command Line:*

Add width= and/or height= followed by their integer values to the parser settings string to indicate the width and/or the height.

#### Scaling

Determines the method by which the scaling is done. There are three available methods:

* Isotropic with Padding: with this method the width and the height are scaled according to the smaller scaling ratio, and the dimension with the larger scaling ratio is padded with zeros. This is the default method.
* Isotropic with Cropping: with this method the width and the height are scaled according to the larger scaling ratio, and the dimension with the smaller scaling ratio is cropped.
* Anisotropic: with this method, each dimension is scaled according to its own scaling ratio.

*On the Command Line:*

Isotropic with padding is the default scaling method. To use a different method, add scaling=IsoCrop to the parser settings string to indicate isotropic scaling with cropping, or scaling=Aniso to indicate anisotropic scaling.

#### Pixel Components

Indicate which components of the image pixels to use. By default, the red, green and blue components are used and the alpha component is not used.

*On the Command Line:*

* Add red=-, green=- or blue=- to the parser settings string to indicate not using the red, green or blue component (respectively)
* Add alpha=+ to the parser settings string to indicate using the alpha component

With the default arguments, the parser creates a feature vector with 256x256x3 features.

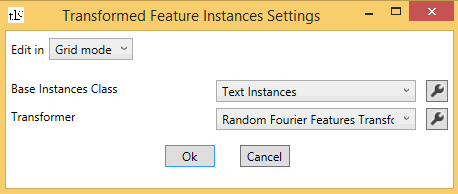
## Transformed Feature Instances

To apply feature transformations to data, choose Transformed Feature Instances in the [Instances Type](#learning_modes_train_mode_htm_in_1141) drop list in the experiment box in the left pane of the TLC GUI.

*On the Command Line:*

Use the command-line argument /inst TransformInstances to specify text format.

This is the dialog box for the settings of Transformed Feature Instances:



### Base Instances Class

The base instances class indicates how the input is read from the dataset file. Click on the image001.png to specify the Instances settings.

*On the Command Line:*

* Add inst= followed by the Instances class name to the IPS string
* Pass the IPS string of the base Instances class to the program after the class name, in a pair of curly brackets

### Transformer

Indicate which transform to apply to the input. In addition to all the TLC normalizers, there are two more transforms available: Random Fourier Features and PCA projection.

*On the Command Line:*

* Add trans= followed by the transformer class name to the IPS string
* Pass the transformer settings string in the format parameter1=value1 parameter2=value2 ... to the program after the transformer name, in a pair of curly brackets

### Example

*On the Command Line:*

* To run Logistic Regression on breast-cancer.txt using the Random Fourier Features transformer in cross-validation mode, write:  
  TL.exe /cl LR Samples\breast-cancer.txt /inst=TransformInstances{trans=RandomFourierFeaturesTransformer{D=500 kernel=GaussianRandom{g=0.1} rng=1}}
* To run Linear SVM on breast-cancer.txt using the PCA transformer in train mode, write:  
  TL.exe /cl LinearSVM Samples\breast-cancer.txt /inst=TransformInstances {trans=pca {rank=3 oversamling=3 }}

## Sample datasets

These are some sample datasets that can be downloaded from [\\cloudmltlc\tlc\Samples](file://cloudmltlc/tlc/Samples).

**Breast-cancer.txt**: In this dataset each instance contains some information (9 features) about a tumor, where the label is 0/1 for benign/malignant. The features are tab-separated. The format of this file is the standard, and thus the IPS string is empty.

**SmartMatch\Instances-TrainRelevance-named.txt**: This dataset contains information about web-search relevance (the labels are binary, for relevant/not relevant). The file contains a header line with the feature names. Each instance has a name, a binary label and 147 features, tab-separated. Each name begins with ‘\_’, thus the IPS string is header=+.

**UnitTest\MSM-sparse-sample-train.txt**: This dataset contains information about web-search relevance, in sparse representation. Each line begins with an instance name (beginning with ‘\_’), a binary label, the total number of features (147) followed by index:value pairs for non-zero features, tab-separated. Since the names of the instances begin with ‘\_’ and the dimension is specified in every line, the IPS string is empty.

**Scope\MSMrel.txt**: This file contains the same dataset as SmartMatch\Instances-TrainRelevance-named.txt, with two differences; There is no header, and the name of each instance is split into the first two columns, and does not begin with ‘\_’. The IPS string for this file is name=0,1.

**UCI\adult.train** and **UCI\adult.test**: In these datasets each instance contains some information about a person, where the label is 0 if that person earns less than 50,000 per year and 0 otherwise. The files contain a header line. The labels are in column 14 and columns 1,3,5-9 and 13 have categorical features. The features are comma-separated. The IPS string for this file is header=+ label=14 cat=1,3,5-9,13 sep=,.

**AmznKaggleSparse\train.csv** and **AmznKaggleSparse\test.csv**: These datasets come from a Kaggle Challenge. They contain historical data collected by Amazon in 2010 and 2011. Each instance contains information about one employee’s role in the company, and the label is whether this employee was allowed or denied access to certain resources. The files contain a header line. Each line contains the label in the first column, followed by 8 categorical features, followed by a name feature. The features are comma-separated. The IPS string for this dataset is header=+ sep=comma cat=1-8 name=9.

**Note:** The categorical features in this dataset have a large number of categories, so converting it into standard format results in extremely sparse feature vectors.

**pclick\pclick-train.txt**, **pclick\pclick-test.txt** and **pclick\CTR.25.txt**: These datasets contain click-through information for web searches. The first two files contain a header line, and two attribute features in the last columns (85 and 86). The last file contains the data in sparse format, with the attribute features omitted. Each line contains the dimension, so there is no need to specify it in the IPS string. The IPS string for the last file is therefore empty, and for the first two files it is header=+ attr=85,86.

**Note:** These datasets are very unbalanced; CTR.25.txt contains approximately 12,000 positive instances and approximately 850,000 negative instances. pclick-train.txt and pclick-test.txt each contain approximately 2,000 positive instances and approximately 96,000 negative instances.

**MNIST\Train-28x28.txt**, **MNIST\Test-28x28.txt**, **MNIST\Train-29x29.txt** and **MNIST\Test-29x29.txt**: These datasets contain images of hand written digits. Each image is encoded as a 28x28 or 29x29 grid of pixels, and each pixel has a value between 0 and 255 indicating the darkness of that pixel. The labels are the digits 0,…,9. The labels are in the first column, and therefore the IPS string for these datasets is empty.

# Output Model Formats

TLC supports the following different output formats for trained models:

## TLC format (binary)

This is the default format, and all algorithms can create this format. This is also the format that TLC uses to load the model for evaluation (in test mode). Starting with the July 2014 release, TLC predictor serialization is versioned, which means that saved models will be loadable in future versions of TLC as well. See [here](#tlc_api_tutorial_using_existing__7646) for details on how to load TLC predictors.

Models saved in this format are saved in a zip directory. One folder in this zip contains the predictor, and another folder contains the information about data pre-processing, such as [feature handlers](#input_data_format_text_input_for_140) and [label mapping](#input_data_format_text_input_for_1020).

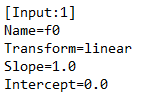
## .ini format

The .ini format corresponds to Bing’s ranker .ini format, and should be readable by Bing’s model evaluators. The algorithms that can output the .ini format are FastRank and FastTree, the linear binary classification algorithms (Averaged Perceptron, Logistic Regression and Linear SVM), SGD and Poisson Regression, and Neural Networks (binary, multi-class and regression).

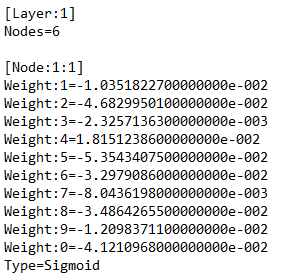
The .ini files produced by the Neural Networks algorithm begin with the lines

NN ini1

Followed by information about each neuron in the input layer:



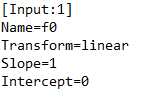
The information about all nodes in the other layers follows in the following format:



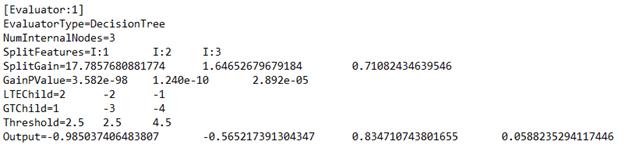
FastRank, FastTree the linear binary classification algorithms (Averaged Perceptron, Logistic Regression and Linear SVM), SGD and Poisson Regression all output the .ini file in the same format. They begin with the following lines:

FR ini1

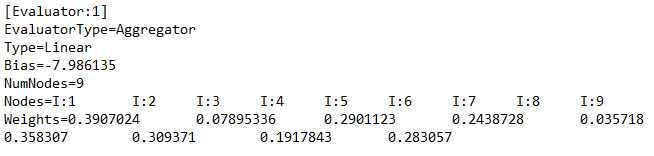
The number of evaluators for FastRank is the number of trees created + 2, and for the rest of the algorithms it is 2. The information about the inputs is the same as in the Neural Networks format:



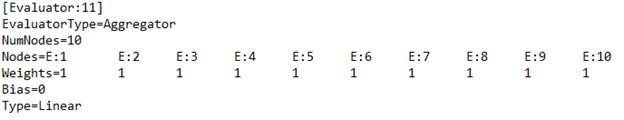
In the FastRank and FastTree .ini files, the description of the trees follows:



The last two evaluators describe the computation of the uncalibrated results, and the calibrated results. The uncalibrated results description is in the following format:

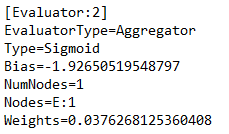


Or for FastRank and FastTree:



The “I” next to each node number indicates this is an input node, and the “E” indicates it is an evaluator node. In FastRank and FastTree ranking, all the weights of this evaluator are always 1, and it computes the score by which the ordering should be done (the sum of the values in the trees).

The calibration information (only for the binary classification algorithms and Poisson Regression) is given in the following format:



The only calibrator that is described in the .ini file is Sigmoid. The calibrated probability given the uncalibrated output image013.png is

image015.png.

If a different calibrator was used, the file only contains an evaluator for the uncalibrated data, with an extra section in the FastRank .ini file:

FR calibration

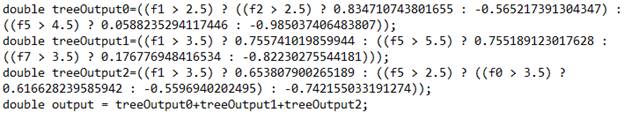
**Note:** For algorithms that support normalization, if normalization is used the .ini file is created but only contains the message:

normalization ini

## Code format

This format produces the model as C++ code. The algorithms that can output the model in code format are FastRank and the linear classification algorithms (Averaged Perceptron, Logistic Regression and Linear SVM).

The following example illustrates the code generated for FastRank:



The following example illustrates the code generated for Logistic Regression:

LR code

The following example illustrates the code generated for Averaged Perceptron or Linear SVM:

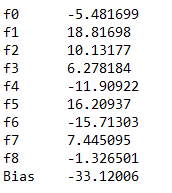
AP SVM code

**Note:** For algorithms that support normalization, the code generated does not take the normalization into account. In other words, to use the code correctly, the data should be normalized first using the same normalization that was used for training.

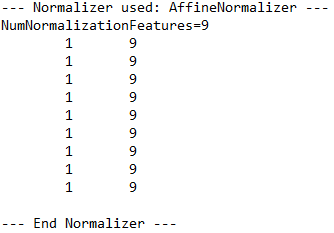
## Text format

This format is intended to be a human-readable text format. The text format output by each algorithm is different. Bramble, MLTK.HNN (binary and multi-class), LDSVM, one-vs-all, pairwise coupling and Poisson Regression currently do not have standardized text format. The following examples illustrate the text format of each algorithm.

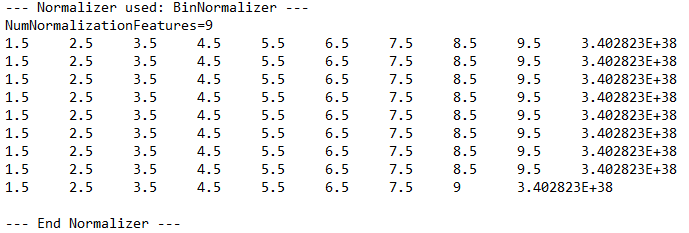
Logistic Regression (binary and multi-class), Averaged Perceptron, Linear SVM and SGD output the weights of the weight vector:



(In Logistic Regression the bias is called intercept, and it appears first instead of last). If normalization was with the linear learners, the normalizer is also described. In the case of [min-max normalizer](#ml_basics_feature_normalization__9237) and [Gaussian normalizer](#ml_basics_feature_normalization__8171) the offsets and scaling factors of each feature are described in the first and second column respectively. If the min-max normalizer is used with the [Fix Zero](#ml_basics_feature_normalization__9237) option, the offsets are 0, and the scaling factor is the maximum value of the original (un-normalized) feature. Otherwise, the offset is equal to the minimum value of the original feature and the scaling factor is equal to the range (maximum value - minimum value) of the original feature. As an example, here is the normalizer description for the [breast-cancer](#input_data_format_sample_dataset_5469) dataset, where all features have values between 1 and 10.

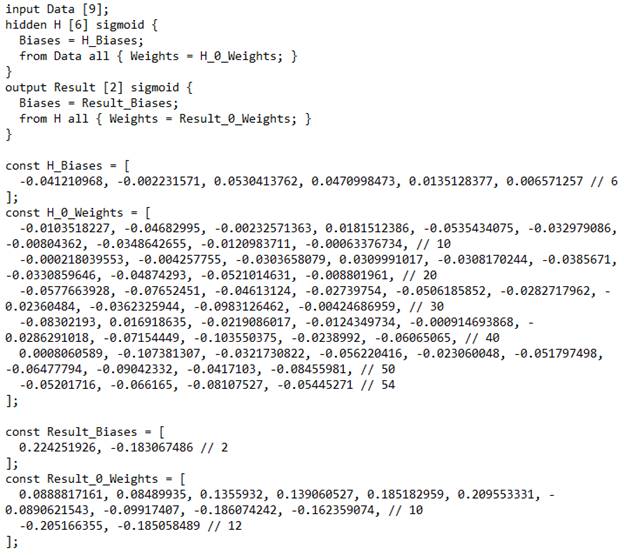


and in the case of [binning normalizer](#ml_basics_feature_normalization__9206) the upper bounds of the bins are described:



The text file generated by FastRank (binary classification, regression and ranking) is identical to the .ini file.

The text file generated by the Neural Networks algorithm (binary, multi-class and regression), includes information about the architecture of the net (including the weights on the edges). The architecture of the net is described in the same format as the initial network structure that the algorithm takes as input. The syntax is described in the [Net# user manual](#net_sharp_(cloudml_neural_networ_6785). Here is an example:



The text file of the Parallel Ensemble algorithm (classification, regression and ranking) contains the text description of each algorithm in the format of the base algorithm.

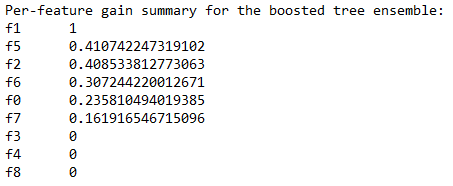
The text file of the multi-class image patching algorithm contains the text descritpion of the base algorithm.

## Model summary

All algorithms can output a model summary, though for some algorithms the summary does not contain any information. These are the Neural Networks algorithm (binary, multi-class and regression), Bramble, MLTK.HNN (binary and multi-class), LDSVM, multi-class image patching, one-vs-all, pairwise coupling and Poisson Regression.

The model summary of the linear algorithms (binary and multi-class Logistic Regression, Averaged Perceptron, Linear SVM and SGD) is identical to the text format.

The model summary of FastRank (classification, regression and ranking) contains the per-feature gain summary for the weighted tree ensemble (ordered from highest to lowest):

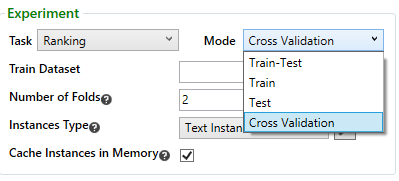


The Parallel Ensemble algorithm summary (classification, regression and ranking) contains a separate summary of each model.

# Specifying the learning mode

TLC provides various ways to train and test a model. These ways are referred to as *learning modes* and are discussed in the following sections.

Specify the learning mode by choosing it from the “mode” drop list on the top of the experiment box in the left pane of the TLC GUI.



In every mode there are some settings to specify. Some settings are displayed in the “Advanced Options” box in the left pane. By default, these options are collapsed for simplicity. They can be toggled by pressing Ctrl + T.

Options that have default values, as well as values entered by the user for some rarely-changed fields, such as run mode, HPC cluster and random seed, are saved in the registry, and remain the same the next time TLC is opened.

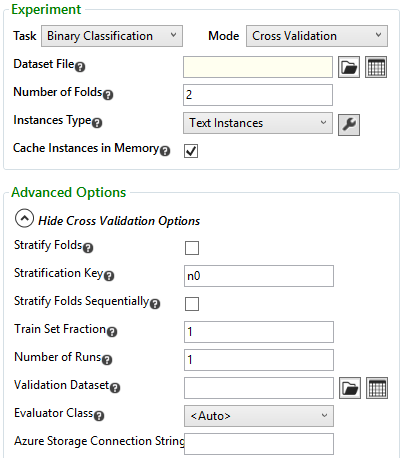
*On the Command Line:*

Use the command-line argument /c <Train|TrainTest|Test|CV>.

## Cross validation mode

Cross validation is a technique used for training and testing a model when there is only one dataset. The dataset is partitioned into image002.png parts (image002.png is specified by the user) called *folds*. Each fold, in turn, is used as a test set, where the rest of the data is used as a training set. The result is image002.png separate models. The metrics for each model are reported separately, and so is the average of each metric on all models.

These are the options that can be specified in cross validation mode:



*On the Command Line:*

Cross validation is the default mode, so no arguments need to be specified.

### Train Dataset

Every learning mode except for test mode uses a training set to train the model. This dataset is specified in the “Train Dataset” text box.

*On the Command Line:*

The file name is passed to the program after the learning mode with no additional arguments.

### Number of Folds

The number of folds determines the number of parts the dataset is partitioned into. The default value is 2.

*On the Command Line:*

Use the command-line argument /k followed by the number of folds to pass it to the program.

### Instances Type

Indicates which class that implements the Instances interface to use to [parse the data](#input_data_format_htm). The default value is Text Instances.

*On the Command Line:*

Use the command-line argument /inst followed by the Instances Type class name to pass the data format to the program.

### Cache Instances in Memory

This check box is checked by default, and it indicates that all the data should be read into the memory even if the algorithm uses one instance at a time.

*On the Command Line:*

Use the command-line argument /cacheinst- to disable data caching.

### Stratify Folds, Stratification Key, Stratify Folds Sequentially

Sometimes there is a need to specify which examples should not be separated into different folds. Take, for instance, the ranking problem, where instances have a “query” feature, and a “url” feature. Instances that have the same query value should always be in the same fold (otherwise the algorithm “cheats” by seeing examples for same query). In this case, check the “Stratify Folds” check box, and specify which instances should not be separated by specifying one or more key features in the “Stratification Key” box. Features that can be used as keys are name features, or attribute features. These are specified by n (for name features) or a (for attribute features), followed by an integer, which is the 0-based index of the name (or attribute) features, as indicated in the [Name Columns (or Attribute Columns)](#input_data_format_text_input_for_1375) text box. If there is more than one key feature, list them separated by commas. For example, if the name features are in columns 3, 5, 9 and 15 and the attribute features are in columns 17 and 23, then if n3,a1is specified as the stratification key, the instances will be stratified by the values in columns 15 and 23 (the 4th name feature column and the 2nd attribute feature column).

The default value for the stratification key is n0 , i.e. the first name feature.

*On the Command Line:*

* Use the command-line argument /strat to indicate stratification keys should be used
* Use the command-line argument /stratkey followed by the stratification keys as explained above
* If /strat is specified but /stratkey is not, the first name feature is used as the stratification key

By default, partitioning the dataset into folds is done by randomizing the order of the instances, and then assigning instances image008.png through image010.png  to the image012.png’th fold (image014.png is the total number of instances). Checking the “Stratify Folds Sequentially” check box, specifies that the order of instances should not be randomized before partitioning into folds.

*On the Command Line:*

Use the command-line argument /foldseq+ to indicate stratifying with no randomization.

### Train Set Fraction

The default value for this option is 1. If a fraction less than 1 is specified, the algorithm only uses this fraction of the training set to learn. This option is useful for generating *learning curves*, by learning multiple times with an increasing fraction of the learning set, and plotting the accuracy vs. the fraction of the dataset used.

*On the Command Line:*

Use the command-line argument /tp followed by a number between 0 and 1 to pass it to the program.

### Validation Dataset

In some algorithms (FastRank, MLTK.HNN), it is possible to specify a file containing an extra dataset. This dataset is used to self-correct during training.

*On the Command Line:*

Use the command-line argument /valid followed by the file name to pass it to the program.

### Evaluator Class

To evaluate the models using custom metrics, choose the evaluator class from the drop list, and click the wrench.png button next to it to specify the settings. Evaluator classes must inherit from the abstract class Tester.

*On the Command Line:*

* Use the command-line argument /ev followed by the class name to pass a custom evaluator to the program
* The evaluator settings are passed in the evaluator settings string after the evaluator name. the evaluator settings string is in the format { parameter1=value1 parameter2=value2 … }

### Number of Runs

Specifies the number of times the dataset should be partitioned into folds. If this number is image017.png and the number of folds is image002.png, the algorithm generates image019.png models. The default value is 1.

*On the Command Line:*

Use the command-line argument /nr followed by the number of runs to pass it to the program.

### Azure Storage Connection String

TLC can run locally using datasets located on Azure, and saving any results on Azure. To use datasets located on Azure, enter the Azure storage connection string in this text box. The format of the string is

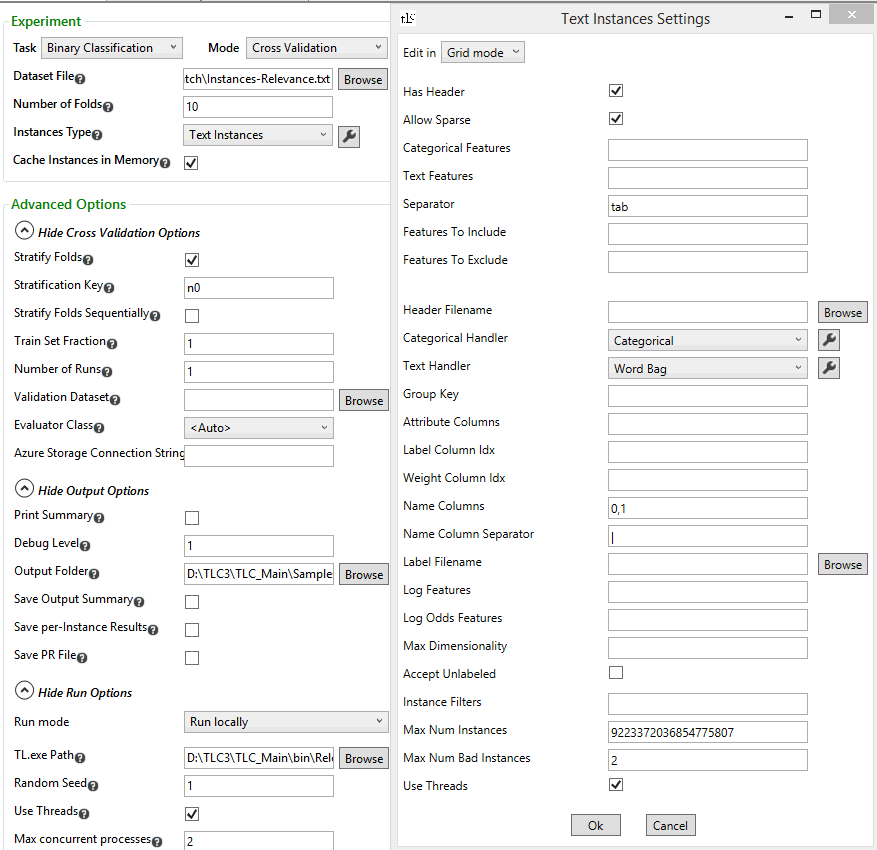
DefaultEndpointsProtocol=<http|https>;AccountName=<Name>;AccountKey=<Key>

*On the Command Line:*

Use the command-line argument /azs followed by the connection string for the Azure storage account to use to pass it to the program.

### Example

To learn using cross validation with 10 folds, from a file that contains a header line and name features in the first and second columns, stratifying the instances by the first name feature, enter these options as follows:



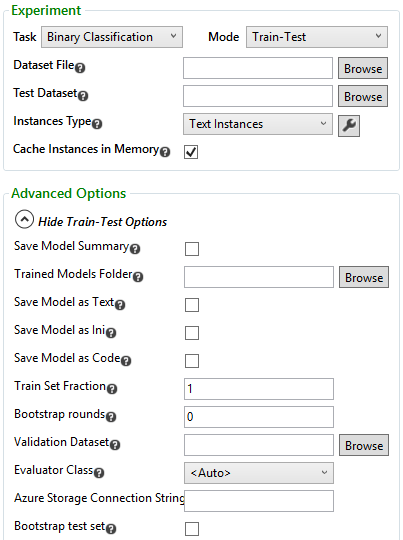
*On the Command Line:*

TL.exe samples\SmartMatch\Instances-Relevance.txt /inst TextInstances { header=+ name=0,1 } /strat /stratkey n0 /k 10

## Train-test mode

When there are two separate files containing labeled data, run TLC in train-test mode. In this mode the model is trained using the data in the training file, and then tested on the data in the test file.

These are the options that can be specified in train-test mode:



*On the Command Line:*

To run in train-test mode, use the command-line argument /c TrainTest.

### Train Dataset

Every learning mode except for test mode uses a training set to train the model. This dataset is specified in the “Train Dataset” text box.

*On the Command Line:*

The file name is passed to the program after the learning mode with no additional arguments.

### Test Dataset

The name of the file that contains the data to test the model on.

*On the Command Line:*

Use the command-line argument /test followed by the file name to pass it to the program.

### Instances Type

Indicates which class that implements the Instances interface to use to [parse the data](#input_data_format_htm). The default value is Text Instances.

*On the Command Line:*

Use the command-line argument /inst followed by the Instances Type class name to pass the data format to the program.

### Cache Instances in Memory

This check box is checked by default, and it indicates that all the data should be read into the memory even if the algorithm uses one instance at a time.

*On the Command Line:*

Use the command-line argument /cacheinst- to disable data caching.

### Trained Model Folder

Specify the folder where TLC generates the model files.

*On the Command Line:*

The full path of the [model file name](#learning_modes_train_mode_htm_sa_1226) must be specified.

### Save Model Summary, Save Model as Text/Ini/Code

Check the check boxes of the formats the model should be saved in. If one or more format boxes are checked, the trained model folder must be specified. If the trained model folder is specified, the model is also saved in binary format (whether other formats are specified or not). The names of the files containing the requested formats are automatically determined as \*.summary.txt, \*.model.txt, \*.model.ini, \*.model.cs and \*.model respectively, where \* is an automatically generated number for the algorithm that produced this model. Only a [folder to save these files](#learning_modes_train_mode_htm_tr_5991) needs to be specified.

*On the Command Line:*

* /ms followed by the file name to save the model summary
* /mt followed by the file name to save the model in text format
* /mi followed by the file name to save the model in ini format
* /mc followed by the file name to save the model in code format
* /m followed by the file name to save the model in binary format

**Note:** Not all algorithms support all the formats. Algorithms that do not support .ini or text format, save the model summary instead. Algorithms that do not support code format, generate a warning and do not create the file. See the next sections for details about supported formats.

### Train Set Fraction

The default value for this option is 1. If a fraction less than 1 is specified, the algorithm only uses this fraction of the training set to learn. This option is useful for generating *learning curves*, by learning multiple times with an increasing fraction of the learning set, and plotting the accuracy vs. the fraction of the dataset used.

*On the Command Line:*

Use the command-line argument /tp followed by a number between 0 and 1 to pass it to the program.

### Bootstrap rounds, Bootstrap test set

The value in this text box indicates the number of additional passes through the training set should be done in the training phase.

If the Bootstrap test set is checked, the test set is resampled in every bootstrapping round.

*On the Command Line:*

* Use the command-line argument /bs followed by the number of bootstrapping rounds to pass it to the program
* Use the command-line argument /bstest+ to indicate resampling the test set during the bootstrapping rounds

### Validation Dataset

In some algorithms (FastRank, MLTK.HNN), it is possible to specify a file containing an extra dataset. This dataset is used to self-correct during training.

*On the Command Line:*

Use the command-line argument /valid followed by the file name to pass it to the program.

### Evaluator Class

To evaluate the models using custom metrics, choose the evaluator class from the drop list, and click the wrench.png button next to it to specify the settings. Evaluator classes must inherit from the abstract class Tester.

*On the Command Line:*

* Use the command-line argument /ev followed by the class name to pass a custom evaluator to the program
* The evaluator settings are passed in the evaluator settings string after the evaluator name. the evaluator settings string is in the format { parameter1=value1 parameter2=value2 … }

### Azure Storage Connection String

TLC can run locally using datasets located on Azure, and saving any results on Azure. To use datasets located on Azure, enter the Azure storage connection string in this text box. The format of the string is

DefaultEndpointsProtocol=<http|https>;AccountName=<Name>;AccountKey=<Key>

*On the Command Line:*

Use the command-line argument /azs followed by the connection string for the Azure storage account to use to pass it to the program.

### Test Instance Settings

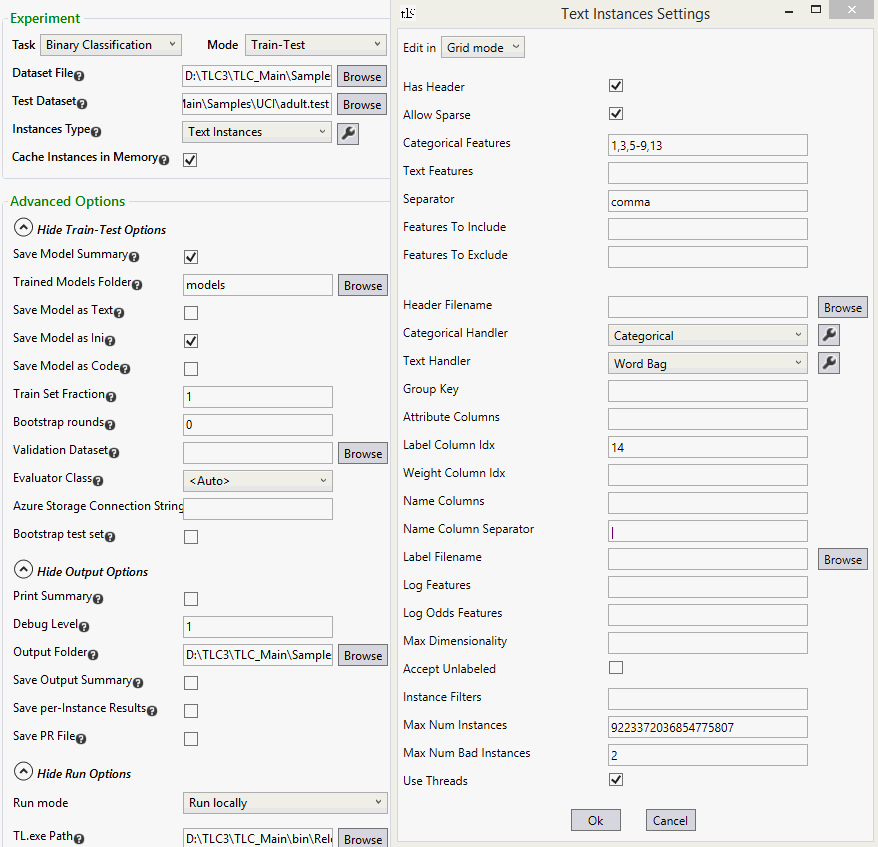
Contains the IPS string for the test file, as explained in section ‎III. This string only needs to be specified if instances in the test set need to be parsed differently than training instances.

*On the Command Line:*

Use the command-line argument /instsettest followed by the test IPS string in a pair of curly brackets to pass it to the program.

### Example

To learn using the train-test mode, from a training file with a header line, the labels in column 14, categorical features in columns 1, 3, 5 thru 9 and 13, separated by commas and a test file in the same format, and to save the model summary and in .ini format in a folder named “models”, enter the options as follows:



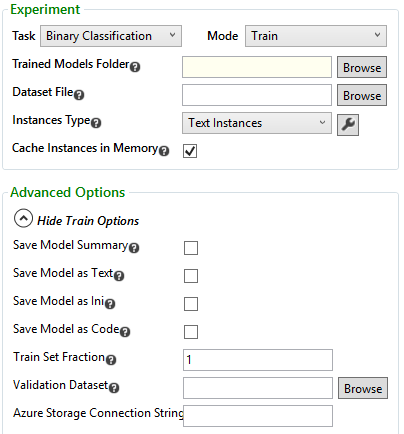
*On the Command Line:*

TL.exe /c TrainTest Samples\UCI\adult.train /inst TextInstances { header=+ label=14 cat=1,3,5-9,13 sep=, } /ms models\0.model.summary.txt /mi models\0.model.ini /m models\0.model /test Samples\UCI\adult.test

## Train mode

The train mode is used to train a model and save it for future use.

These are the options that can be specified in train mode:



*On the Command Line:*

To run in train mode, use the command-line argument /c Train.

### Trained Model Folder

Specify the folder where TLC generates the model files.

*On the Command Line:*

The full path of the [model file name](#learning_modes_train_mode_htm_sa_1226) must be specified.

### Train Dataset

Every learning mode except for test mode uses a training set to train the model. This dataset is specified in the “Train Dataset” text box.

*On the Command Line:*

The file name is passed to the program after the learning mode with no additional arguments.

### Instances Type

Indicates which class that implements the Instances interface to use to [parse the data](#input_data_format_htm). The default value is Text Instances.

*On the Command Line:*

Use the command-line argument /inst followed by the Instances Type class name to pass the data format to the program.

### Cache Instances in Memory

This check box is checked by default, and it indicates that all the data should be read into the memory even if the algorithm uses one instance at a time.

*On the Command Line:*

Use the command-line argument /cacheinst- to disable data caching.

### Save Model Summary, Save Model as Text/Ini/Code

Check the check boxes of the formats the model should be saved in. If one or more format boxes are checked, the trained model folder must be specified. If the trained model folder is specified, the model is also saved in binary format (whether other formats are specified or not). The names of the files containing the requested formats are automatically determined as \*.summary.txt, \*.model.txt, \*.model.ini, \*.model.cs and \*.model respectively, where \* is an automatically generated number for the algorithm that produced this model. Only a [folder to save these files](#learning_modes_train_mode_htm_tr_5991) needs to be specified.

*On the Command Line:*

* /ms followed by the file name to save the model summary
* /mt followed by the file name to save the model in text format
* /mi followed by the file name to save the model in ini format
* /mc followed by the file name to save the model in code format
* /m followed by the file name to save the model in binary format

**Note:** Not all algorithms support all the formats. Algorithms that do not support .ini or text format, save the model summary instead. Algorithms that do not support code format, generate a warning and do not create the file. See the next sections for details about supported formats.

### Train Set Fraction

The default value for this option is 1. If a fraction less than 1 is specified, the algorithm only uses this fraction of the training set to learn. This option is useful for generating *learning curves*, by learning multiple times with an increasing fraction of the learning set, and plotting the accuracy vs. the fraction of the dataset used.

*On the Command Line:*

Use the command-line argument /tp followed by a number between 0 and 1 to pass it to the program.

### Validation Dataset

In some algorithms (FastRank, MLTK.HNN), it is possible to specify a file containing an extra dataset. This dataset is used to self-correct during training.

*On the Command Line:*

Use the command-line argument /valid followed by the file name to pass it to the program.

### Azure Storage Connection String

TLC can run locally using datasets located on Azure, and saving any results on Azure. To use datasets located on Azure, enter the Azure storage connection string in this text box. The format of the string is

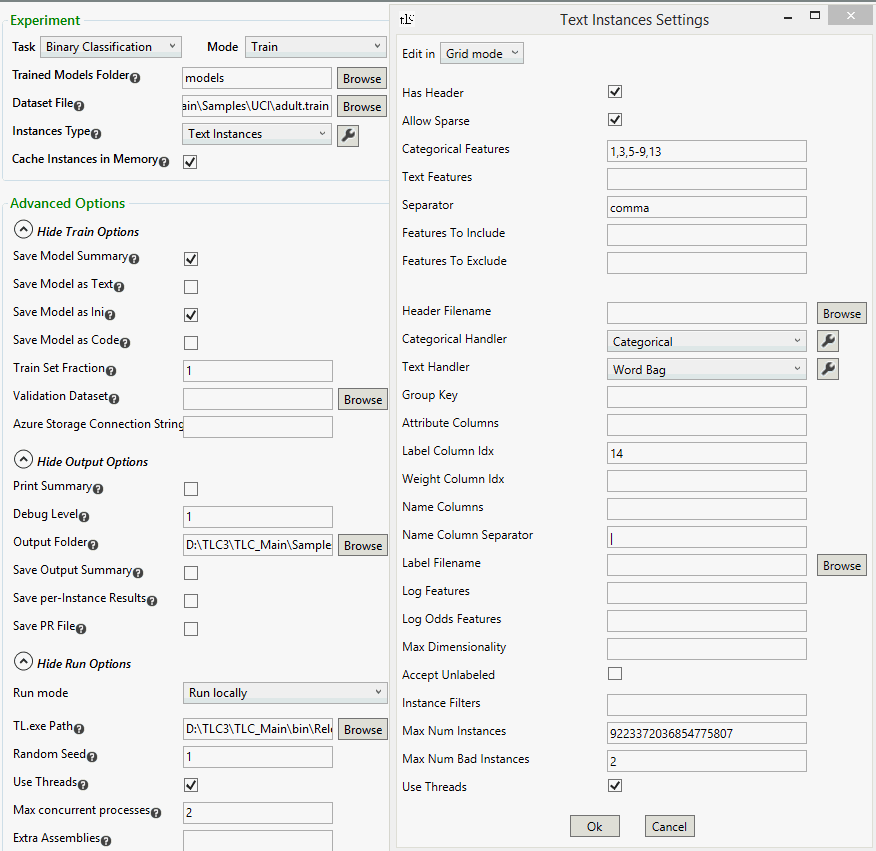
DefaultEndpointsProtocol=<http|https>;AccountName=<Name>;AccountKey=<Key>

*On the Command Line:*

Use the command-line argument /azs followed by the connection string for the Azure storage account to use to pass it to the program.

### Example

To train a model using the train mode, from a training file with a header line, the labels in column 14, categorical features in columns 1, 3, 5 thru 9 and 13, separated by commas and a test file in the same format, and to save the model summary and in .ini format in a folder named “models”, enter the options as follows:



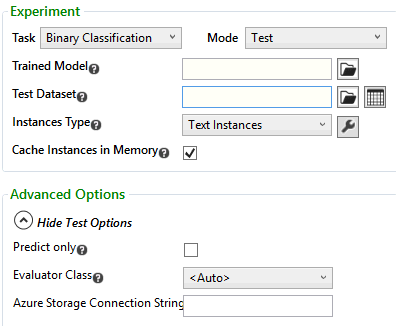
*On the Command Line:*

TL.exe /c Train Samples\UCI\adult.train /inst TextInstances { header=+ label=14 cat=1,3,5-9,13 sep=, } /ms models\0.model.summary.txt /mi models\0.model.ini /m models\0.model

## Test mode

The test mode is used to test a model that was saved by a run in train (or train-test) mode.

These are the options that can be specified in Test mode:



*On the Command Line:*

To run in test mode, use the command-line argument /c Test.

### Trained Model

Specifies the name of the file containing the model that should be used.

*On the Command Line:*

Use the command-line argument /m followed by the file name to pass it to the program.

### Test Dataset

The test set is specified in the "Test Dataset" text box.

*On the Command Line:*

Use the command-line argument /test followed by the test file name to pass it to the program.

### Instances Type

Indicates which class that implements the Instances interface to use to [parse the data](#input_data_format_htm). The default value is Text Instances.

*On the Command Line:*

Use the command-line argument /inst followed by the Instances Type class name to pass the data format to the program.

### Predict Only

In order to score/predict only instead of testing, the label column is not needed. If that is the case, “Predictor only” check box should be checked. If the label column exists, it is ignored.

*On the Command Line:*

Use the command-line argument /po=+ to indicate predicting instead of testing.

### Evaluator Class

To evaluate the models using custom metrics, choose the evaluator class from the drop list, and click the wrench.png button next to it to specify the settings. Evaluator classes must inherit from the abstract class Tester.

*On the Command Line:*

* Use the command-line argument /ev followed by the class name to pass a custom evaluator to the program
* The evaluator settings are passed in the evaluator settings string after the evaluator name. the evaluator settings string is in the format { parameter1=value1 parameter2=value2 … }

### Azure Storage Connection String

TLC can run locally using datasets located on Azure, and saving any results on Azure. To use datasets located on Azure, enter the Azure storage connection string in this text box. The format of the string is

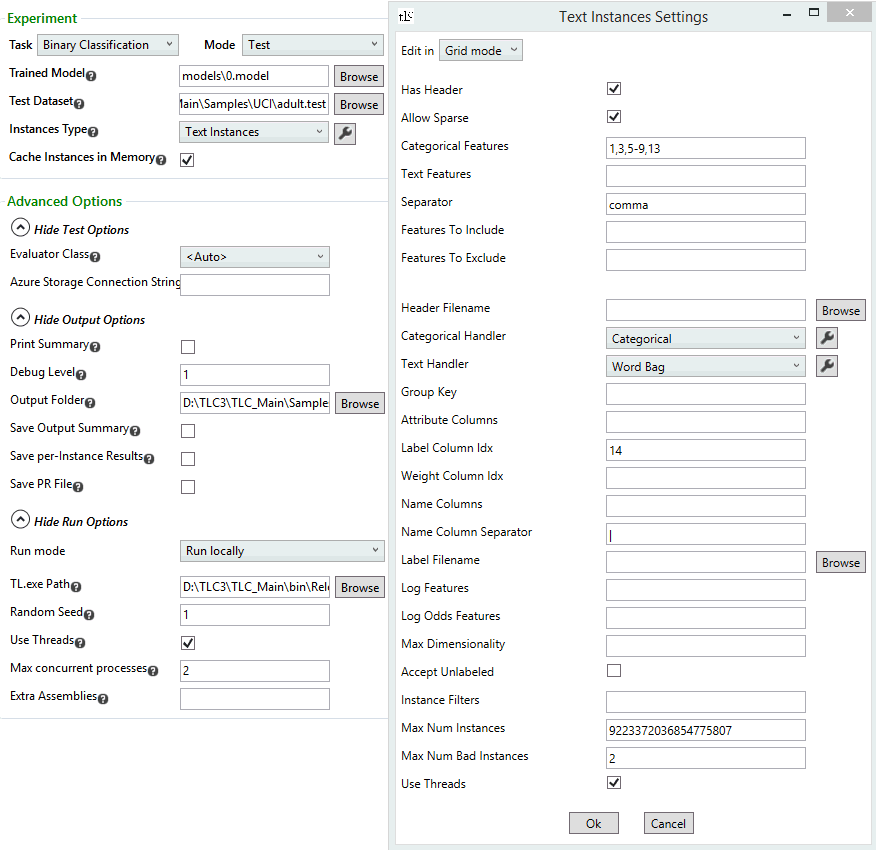
DefaultEndpointsProtocol=<http|https>;AccountName=<Name>;AccountKey=<Key>

*On the Command Line:*

Use the command-line argument /azs followed by the connection string for the Azure storage account to use to pass it to the program.

### Example

To test a model saved in the file models\0.model on a test file with a header line, the labels in column 14, categorical features in columns 1, 3, 5 thru 9 and 13, separated by commas, enter the options as follows:



*On the Command Line:*

TL.exe /c Test Samples\UCI\adult.test /inst TextInstances { header=+ label=14 cat=1,3,5-9,13 sep=, } /m models\0.model

The optional [/o](#choosing_the_output_options_htm__5399) argument, followed by a file name, specifies a file where the labels generated by the model for each instance are saved.

# Other TLC modes

TLC can run in two additional modes from the command line:

## CreateInstances mode

The CreateInstances mode is used to convert data format, normalize it, or add per-feature statistics. To run TLC in this mode, use the command-line argument /c CreateInstances. CreateInstances mode can create files in two formats: text and binary ([TLCBIN format](#input_data_format_binary_input_f_7440)). To create a [text file](#other_tlc_modes_createinstances__1350) add /writer TextInstanceWriter (this is the default writer, so it can be omitted) to the command-line arguments, and to create a [TLC binary file](#other_tlc_modes_createinstances__3452) add /writer BinaryInstanceWriter to the command-line arguments. In each case, the command may be followed by a *writer settings string* in the format { parameter1=value1 parameter2=value2 … }.

### Converting data into standard text format

Data with categorical or text features requires an additional pass for category mapping. Thus, it is recommended to convert the data once to standard format, and then use a TLC BIN file as the data file with default settings (without an IPS string).

To convert data to standard format, run TLC in CreateInstances mode, adding /inst TextInstances followed by the IPS string in a pair of curly brackets to the command-line arguments. The file created contains, instead of each [categorical or text feature](#input_data_format_text_input_for_9140), a set of numeric features. For example:

TL.exe /c CreateInstances DataFileName.train /test DataFileName.test /inst TextInstances { header=+ label=14 cat=1,3,5-9,13 sep=, }

The name of the file created is the same as the original input file, with the additional suffix .tlc.

The TextInstanceWriter supports the option to include a [weight column](#input_data_format_text_input_for_5430) after the label column. If the input file has a weight column it is specified in the IPS string, and otherwise the weights are all 1. To include a weight column add weights=+ to the writer settings string.

Conversion to standard text format may result in sparse data, with which feature headings are no longer present. To keep data in dense format, with the default header that has feature names, add dense=+ to the writer settings string. When dense format is requested, the created file contains a header line. If the original file contains feature names, these names are used in the new file as well, and otherwise the feature names are f1, f2 and so on, and the label name is Label.

**Note**: Files in TLCBIN format may also be converted to text format, by using /inst BinaryInstances instead of /inst TextInstances.

### Converting data into binary format

When the input file is very large, it is recommended to convert it to [TLCBIN format](#input_data_format_binary_input_f_7440). This can be done in CreateInstances mode, by adding /writer BinaryInstanceWriter to the command-line arguments.

When the data contains image information, this phase is especially important, since the pixel extraction phase of the data parsing is extremely time consuming. To convert files in [image format](#input_data_format_image_input_fo_8435) to TLCBIN format, add /inst ImageInstances to the command-line arguments.

The file in TLCBIN format created has the same name as the original input file, with the extra suffix .tlcbin.

The user defined parameters that can be added to the writer settings string of the BinaryInstanceWriter are:

* dense=+ to force the output to have a dense format
* ltype=and ftype= followed by number types (e.g R4, R8, U1, I1 etc) to indicate the number types for the labels and the features (respectively). The default values are R4
* lscale= and fscale=followed by values to indicate the scaling factors for the labels and features (respectively). The default values are 1. The scaling can either be done in the writing phase (using the BinaryInstanceWriter) or in the loading phase (using [BinaryInstances](#input_data_format_binary_input_f_7440)).
* stats=+  to indicate collecting per-feature statistics
* blockRows= followed by a non-negative integer to indicate writing the data in blocks of this many rows
* subformat= followed by the sub-format to use in block writing and its settings string in a pair of curly brackets (the options are DenseCol, DenseRow or SparseRow)
* threads= followed by the number of worker compression threads to use
* keepLabel=-, keepName=- or keepAttribute=- to indicate omitting the label, name or attributes (respectively) to the output file

### Data normalization

Using normalized data can provide significant speedup and accuracy benefits. Thus, [normalizing data](#ml_basics_feature_normalization__7408) once saves time by allowing normalization to be turned off for each algorithm run. This option can be used when using either TextInstancesWriter or BinaryInstancesWriter. Normalize data by running CreateInstances mode and adding /norm= followed by the normalizer name to the command-line arguments. The different normalizers are [MinMaxNormalizer](#ml_basics_feature_normalization__9237), [BinNormalizer](#ml_basics_feature_normalization__9206) and [GaussianNormalizer](#ml_basics_feature_normalization__8171).

Data can also be transformed in CreateInstances mode by using [Transformed Feature Instances](#input_data_format_feature_transf_3479).

### Per-feature statistics

The CreateInstances mode can also be used to compute basic statistics for each feature. This is often very useful in debugging and comparing the training dataset and the test dataset as well as assessing data compatibility or feature drift. This option can be used when using either TextInstancesWriter or BinaryInstancesWriter. To create the statistics file, add stats=+ to the writer settings string.

The statistics for each data file are located in a file with the same name, ending with the additional suffix .stat.tsv. The per-feature statistics are mean value, standard deviation, sparsity, minimum value and maximum value.

## FeatureSelection mode

Greedy feature selection is performed by running a specified classifier in either cross-validation or train-test mode iteratively, either adding or subtracting features. The two supported modes are bottom-up (starting with no features, and greedily adding features which give the highest accuracy gains), or top-down (starting with all features, and greedily removing features whose removal doesn’t reduce the accuracy). The bottom-up approach is also called *selection*, and the top-down approach is also called *ablation*.

To run in feature selection mode, use the command-line argument /c FeatureSelection. The name of the input data file is specified with no additional arguments.

The command-line arguments that can be used in this mode are:

### /fsm

/fsm specifies whether to do the selection using cross validation (/fsm CV) or train-test (/fsm TrainTest). If the latter is used, add /test followed by the test file name to the command-line arguments. The default mode is cross validation.

### /fso

/fso specifies the name of the output file for the feature selection output. This option is useful, since the FeatureSelection mode produces a screenful of output, and specifying the output file name filters it down to just the feature-selection-relevant output.

### /fsf

To run feature selection only on a subset of features, add /fsf followed by a comma-separated list of integers to the command-line arguments. The list may contain ranges of indices.

### /fsa

/fsa followed by a settings string indicating other parameter settings. For a complete list of the parameters that can be specified, run TL.exe /c FeatureSelection /fsa ?. The settings string is in the format { parameter1=value1 parameter2=value2 … }.

These are the available user defined parameters:

#### Selection/ablation

To specify selection, add bt=- to the settings string, and to specify ablation, add bt=+ to the settings string. If nothing is specified, the default is selection.

#### Metric

To specify which metric to use to measure the accuracy of the model, add resultIdx= or residx= followed by the index of the metric to the settings string. The index indicating each metric varies according to the task.

For **binary classification**, the indices are

|  |  |
| --- | --- |
| 0 | Accuracy |
| 1 | Positive precision |
| 2 | Positive recall |
| 3 | Negative precision |
| 4 | Negative recall |
| 5 | Log-loss |
| 6 | Log-loss reduction |
| 7 | AUC |

For **multi-class classification**, the indices are

|  |  |
| --- | --- |
| 0 | Accuracy (micro-avg) |
| 1 | Accuracy (macro-avg) |
| 2 | Log-loss |
| 3 | Log-loss reduction |

For **regression**, the indices are

|  |  |
| --- | --- |
| 0 | L1 (avg) |
| 1 | L2 (avg) |
| 2 | RMS (avg) |
| 3 | Loss-fn (avg) |

For **ranking**, the indices are

|  |  |
| --- | --- |
| 0 | NDCG@1 |
| 1 | NDCG@2 |
| 2 | NDCG@3 |
| 3 | DCG@1 |
| 4 | DCG@2 |
| 5 | DCG@3 |

The default metric for binary classification is [AUC](#introduction_classification_meas_1203), for multi-class classification it is [log-loss](#introduction_classification_meas_1072), for regression it is [RMS](#introduction_regression_measurin_6168) and for ranking it is [NDCG@3](#introduction_ranking_measuring_t_1354).

#### IsIncreasePositive

Add IsIncreasePositive= (or pos=) followed by a plus or a minus to the settings string, to indicate whether the metric increase should be positive (for example, for the accuracy metric) or negative (for example, for the log-loss metric). The default value is +.

#### Error tolerance

To specify the error tolerance for continuing to select features, add tolerance= followed by a floating point number to the settings string. The default value is 0. E.g., setting it to 1e-2 allows continuing to select features, even if the accuracy decreases by 0.01.

#### Output prefix

To specify a prefix for every feature selection output line, add fsOutputPrefix= followed by the prefix to the settings string. The default value is “FS: “.

### Examples

The following command-line runs feature selection, and outputs the feature selection relevant output to FSOutput.txt, and all the other output to out.txt:

TL.exe data.txt /c FeatureSelection /fso FSOutput.txt > out.txt

The following command-line runs feature selection using FastRank, with ablation, removing features based on the log-loss metric:

TL.exe data.txt /c FeatureSelection /cl FastRank /fsa {bt=- residx=5 } /fso FSOutput.txt > out.txt

### Other alternatives to FeatureSelection mode

The greedy feature selection functionality described above is quadratic in the number of features, and hence has unreasonable runtime for all but small or low-dimensional datasets. Some faster alternatives exist and are explained below.

#### Using FastRank

Run boosted tree classification or regression (/cl FR or /cl FRR respectively) and look at the feature summary (produced by [/summary](#choosing_the_output_options_htm__7976)). Features with weight 0 are not used by any tree splits, and are likely candidates for removal. As always, do at least some [parameter sweeping](#ml_basics_using_parameter_sweeps_6818).

#### Using logistic regression

Run L1-regularized logistic regression (/cl LR { L1=1 }) that is more aggressive about assigning a weight of 0 to features, and hence identifies features to remove.

#### Manual feature selection

Manually run simple ablations where individual features are knocked out. This can be done by defining features as [attributes](#input_data_format_text_input_for_9658).

# ML Basics

## Choosing the right algorithm

Machine learning problems can generally be split into two main types of problems. In the first, each instance contains a large number of “low level” features. The simplest example for this type of problem is known as “bag of words”. For instance, when the data is composed of documents that need to be classified by topic, a feature is created for every word that appears in the documents, and the value of this feature is the number of occurrences of this word in the document. The second type of problem is when each instance contains “high level”, or engineered features. The most common example for this type of problem is when the instances are query-url pairs, and they have features such as number of clicks and relevance (the former is automatically computed, the value being accumulated over a long period of time, the latter is decided by judges).

Machine learning algorithms can also be split into two main types; algorithms that produce *linear* models, and algorithms that produce *non-linear* models. The linear models (such as Averaged Perceptron, Logistic Regression or Linear SVM) predict the outcome by calculating the inner product of the instance with some *weight vector* (and possibly applying some function to the result). The non-linear models have more complex graph structures, such as regression trees or neural networks.

The rule of thumb is, that when given a “low level” machine learning problem, it is preferable to create a linear model, and when given a “high level” machine learning problem, it is preferable to create a non-linear model. Another rule is, that if there is a high level of interaction between the features, then non-linear models are preferable. High level of interaction between features occurs in “high level” type problems as described above, but also in signal processing problems, such as image or audio recognition.

The Neural Networks algorithm performs best when the result is expected to be a smooth, continuous function. It works well on datasets with “well-behaved” features, e.g., zero mean and unit variance. It also performs well on vision or audio problems (low level features). FastRank, on the other hand, trains very fast, especially when the data has good, high level features. It also has the ability to provide sparse models (and avoid some costly features). FastRank works well in cases where the result is expected to be discontinuous (since the models FastRank produces are piece-wise constant functions). It also works well in cases where the features are not “well-behaved”, without the need to normalize the data and scale the learning rate. FastRank is not suitable for high-dimensional data (~100,000 features or more).

**Tip:** When possible, it is always recommended to run as many experiments as possible in order to find the best model. Experiments may include learning with different learning algorithms, as well as running [parameter sweeps](#ml_basics_using_parameter_sweeps_6818) on each algorithm.

### Examples

The first example is learning with the datasets rcv1.train.tlc and rcv1.test.tlc. Each instance in these datasets is a Reuter’s news segment, and it is labeled 1 if it is in the “corporate acquisitions” category. This example fits the “bag of words” case. The file contains approximately 8,000 instances, each instance having approximately 47,000 features (in sparse format). These are the results of running some classification algorithms on these datasets:

|  |  |  |  |
| --- | --- | --- | --- |
| Algorithm | Accuracy | AUC | Runtime (seconds) |
| Averaged Perceptron | 0.948 | 0.9853 | 59.822 |
| Logistic Regression | 0.9482 | 0.9868 | 64.743 |
| Linear SVM | 0.9425 | 0.9833 | 88.869 |
| FastRank | 0.9031 | 0.9643 | 526 |
| Neural Networks | 0.9481 | 0.9876 | 3503 |

As can be seen from the table above, FastRank and Neural Networks are considerably slower on these datasets (to shorten the runtime of FastRank to a reasonable amount of time, the number of trees created was decreased, at the expense of the accuracy of the results).

The second example is learning with a dataset containing Bing relevance data. The dataset has 100,000 instances, each instance having approximately 2300 features (in sparse format). The labels are 1 if the relevance label is “Perfect”, “Excellent”, “Good” or “Fair”, and 0 if it is “Bad”. These are the results of running some classification algorithms on these datasets:

|  |  |  |  |
| --- | --- | --- | --- |
| Algorithm | Accuracy | AUC | Runtime (seconds) |
| Averaged Perceptron | 0.822 | 0.8269 | 82.112 |
| Logistic Regression | 0.8012 | 0.8258 | 950 |
| Linear SVM | 0.8108 | 0.8533 | 300 |
| FastRank | 0.8486 | 0.9006 | 568 |
| Neural Networks | 0.8251 | 0.8713 | 886 |

As can be seen from the table above, even though the runtime of some of the linear algorithms is better than the runtime of FastRank and Neural Networks, the models learned by FastRank and Neural Networks are generally better than the linear models.

The next example is learning with the MNIST dataset. This dataset contains 50,000 images of hand written digits. Each image is encoded as a 29x29 grid of pixels, and each pixel has a value between 0 and 255 indicating the darkness of that pixel. The labels are the digits 0,…,9. These are the results of running some classification algorithms on this dataset. For algorithms that do not have a multi-class version, the [one-vs-all technique](#introduction_classification_mult_4975) was used.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm | Accuracy | Log-loss | Log-loss reduction | Runtime |
| Averaged Perceptron | 0.9173 | 0.3652 | 84.1297 | 33.009 |
| Logistic Regression | 0.9222 | 0.2956 | 87.1505 | 40.109 |
| Linear SVM | 0.9098 | 0.4078 | 82.2765 | 33.418 |
| FastRank | 0.9685 | 0.1319 | 94.2678 | 147 |
| Neural Networks | 0.9862 | 0.0529 | 97.7018 | 247 |

As can be seen from the table above, while their runtimes are considerably longer, FastRank and Neural Networks perform better on this problem.

The MNIST dataset was also converted into a binary classification problem, by assigning the label 0 to the digits 0-4 and the label 1 to the digits 5-9. These are the results of running some classification algorithms on this dataset as a binary classification problem:

|  |  |  |  |
| --- | --- | --- | --- |
| Algorithm | Accuracy | AUC | Runtime |
| Averaged Perceptron | 0.8596 | 0.9251 | 5.298 |
| Logistic Regression | 0.8635 | 0.9289 | 16.962 |
| Linear SVM | 0.8478 | 0.9151 | 11.236 |
| FastRank | 0.9686 | 0.9945 | 28.123 |
| Neural Networks | 0.9085 | 0.9692 | 183 |

From this example it can also be seen that FastRank and Neural Networks perform better, where FastRank has better results than Neural Networks (this is consistent with the fact that the artificial conversion of this problem into a binary classification problem results in discontinuous data, in the sense that the set of instances with a specific label is discontinuous).

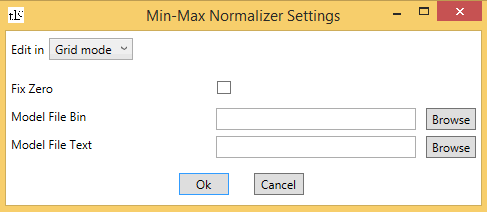
## Feature normalization

In linear classification algorithms instances are viewed as vectors in multi-dimensional space. Since the range of values of raw data varies widely, some objective functions do not work properly without normalization. For example, if one of the features has a broad range of values, the distances between points is governed by this particular feature. Therefore, the range of all features should be normalized so that each feature contributes approximately proportionately to the final distance. This can provide significant speedup and accuracy benefits. In all the linear algorithms in TLC (Logistic Regression, Averaged Perceptron, LinearSVM, SGD), the default is to normalize features before training.

There are a few possible normalizers to choose from, and some normalizers have their own user defined parameters that can be specified by clicking the image001.png button on the right of the normalizer name:

feature normalizer.png

Clicking this button opens a dialog box to specify the relevant parameter values:



Alternatively, the parameter values can be passed using a *normalizer settings string*, by choosing “Text mode” from the drop list on the top of the dialog box  and writing the normalizer settings string in the format parameter1=value1 parameter2=value2 ….

*On the Command Line:*

* For details on how to indicate which normalizer to use on the command line, see the section for each specific algorithm
* The normalizer settings string is passed to the program after the normalizer name, in a pair of curly brackets

### Min-Max normalizer

The min-max normalizer (which is the default normalizer) linearly rescales every feature to the [0,1] or the [-1,1] interval.

* Rescaling to the [0,1] interval is done by shifting the values of each feature so that the minimal value is 0, and then dividing by the new maximal value (which is the difference between the original maximal and minimal values).
* Rescaling to the [-1,1] interval is done by dividing the values of each feature by the maximal absolute value of the feature. This method is useful for preserving the sparsity of a dataset, since 0 values do not change.

The scaling method can be specified by checking the Fix Zero check box for the second method, or unchecking it for the first method.

The trained normalizer can be saved in binary or text format by specifying the file names in the "Model File Bin" and "Model File Text" text boxes. In text format, the file contains the number of features in the first row, followed by one row for each feature, with the offset of the feature, and its scaling factor.

*On the Command Line:*

* Add zero=+ to the normalizer settings string to indicate preserving sparsity
* Add mb= followed by the file name to save the model in binary format
* Add mt= followed by the file name to save the model in text format

### Gaussian normalizer

The Gaussian normalizer rescales the values of each feature to have mean 0 and variance 1. This is done by computing the mean and the variance of each feature, and then, for each instance, subtracting the mean value and dividing by the square root of the variance (the standard deviation).

The trained normalizer can be saved in binary or text format by specifying the file names in the "Model File Bin" and "Model File Text" text boxes.

*On the Command Line:*

* Add mb= followed by the file name to save the model in binary format
* Add mt= followed by the file name to save the model in text format

### Binning normalizer

The binning normalizer creates equi-density bins, and then normalizes every value in the image007.png’th bin to be image007.png divided by the total number of bins.

The number of bins the normalizer users can be defined by the user, and the default number of bins is 1000.

*On the Command Line:*

Add n= followed by the number of bins to the normalizer settings string.

## Feature transformation

When trying to train machine learning models, there is commonly a need to extract or create new features from the given data. This is what feature transformers do.

There are a few possible feature transformers to choose from in TLC, and some transformers have their own user defined parameters that can be specified by clicking the image001.png button on the right of the transformer name. All of the TLC [normalizers](#ml_basics_feature_normalization__7408) are also feature transformers and can be used in the same way as the rest of the transformers.

Feature transformers are accessed by using the [Transformed Feature Instances](#input_data_format_feature_transf_3479) class.

### Random Fourier Features Transformer (RFF Transformer)

This transformer maps the input data to a random low-dimensional feature space. It is useful when data has non-linear features, since the transform is designed so that the inner products of the transformed data are approximately equal to those in the feature space of a user speciﬁed shift-invariant kernel. With this transform, we are able to use linear methods (which are scalable) to approximate more complex kernel SVM models.

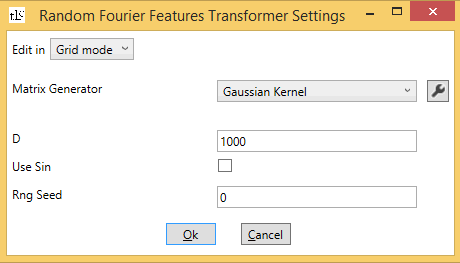
The random Fourier features transformer is based on the paper [Random Features for Large-Scale Kernel Machines](http://pages.cs.wisc.edu/~brecht/papers/07.rah.rec.nips.pdf) by Ali Rahimi and Ben Recht.

**Note:** normalizing the data before transforming it with the RFF transformer can, in most cases, improve results. To do this, specify Transformed Feature Instances as the [base instances class](#input_data_format_feature_transf_3191) in the [Transformed Feature Instances](#input_data_format_feature_transf_3479) settings string, and defining its [transformer](#input_data_format_feature_transf_1971) to be [GaussianNormalizer](#ml_basics_feature_normalization__8171).

*On the Command Line:*

To normalize the data before transforming it, define the IPS string as follows:   
/inst Trans {inst=Trans{trans=GaussianNormalizer} trans=RFF{rng=1 ...}}

Clicking on the image001.png button next to the Random Fourier Features Transformer name opens a dialog box to specify its parameters:



Alternatively, the parameter values can be passed using a *transformer settings string*, by choosing “Text mode” from the drop list on the top of the dialog box  and writing the transformer settings string in the format parameter1=value1 parameter2=value2 ….

*On the Command Line:*

* RFF transformer is specified by adding trans=RFF to the IPS string of Transformed Feature Instances.
* The transformer settings string should follow, in a pair of curly brackets.

#### Matrix Generator

The RFF transformer produces data whose inner product approximates a kernel-dot-product in the original data space. This transform can approximate two kernels: the Gaussian kernel and the Laplacian kernel. Each of these kernels has one user defined parameter, gamma, that can be specified by clicking on the image001.png button next to the kernel name.

*On the Command Line:*

* Add kernel= followed by GaussianRandom or LaplacianRandom to the transformer settings string
* The settings of the kernel are passed to the program by adding { g = value } after the kernel name

#### D

This parameter indicates the dimension to map the data to.

*On the Command Line:*

Add D= followed by its value to the transformer settings string.

#### Use Sin

The RFF transformer maps an instance to a new instance by multiplying the original feature vector by a matrix with D rows, and either applying cosine to each coordinate in the result (with a random bias), or applying cosine and sine to each coordinate in the result. The former is used when the Use Sin check box is unchecked, thus mapping the data into D-dimensional space, and the latter is used when the Use Sin check box is checked, thus mapping the data into 2D-dimensional space.

*On the Command Line:*

Add useSin= followed by + or - to the transformer settings string.

#### Rng Seed

This parameter indicates the random seed that is used to create the transform matrix.

**Note:** it is important to specify a non-zero random seed for the random number generator to improve results.

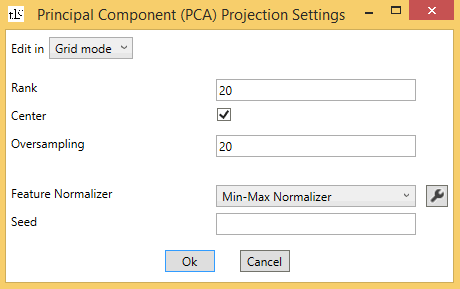
*On the Command Line:*

Add rng= followed by its value to the transformer settings string.

### PCA Projection Transformer

PCA is a dimensionality-reduction transformer which computes the projection of the feature vector to onto a low-rank subspace. Its training is done using the technique described in the paper [Combining Structured and Unstructured Randomness in Large Scale PCA](http://arxiv.org/pdf/1310.6304v2.pdf) by Nikos Karampatziakis and Paul Mineiro, and the paper [Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions](http://arxiv.org/pdf/0909.4061v2.pdf) by N. Halko et al.

Clicking on the image001.png button next to the PCA Projection Transformer name opens a dialog box to specify its parameters:



#### Rank

Indicates the number of components in the PCA. The default value is 20.

*On the Command Line:*

Add rank= followed by the value to the learner settings string.

#### Center

When this check box is checked,  the data is centered to have zero mean. It is checked by default.

*On the Command Line:*

Add center=- to the learner settings string to specify not centering the data.

#### Oversampling

This parameter is used in randomized PCA to control the accuracy. The default value is 20.

*On the Command Line:*

Add o= followed by the value to the learner settings string.

#### Feature Normalizer

By default, the features are normalized using the min-max normalizer. To specify no normalization, choose <None> from the Normalizer drop list. A different normalizer can also be chosen using this drop list.

*On the Command Line:*

* Add norm={} to the learner settings string to indicate no normalization
* Add norm=GaussianNormalizer or norm=BinNormalizer to the learner settings string to indicate normalization using a different normalizer. If a normalizer settings string is needed it follows the normalizer name, in a pair of curly brackets

#### Seed

The seed for the random number generator used by the trainer. If not specified, a random seed is used.

*On the Command Line:*

Add seed= followed by the value to the learner settings string.

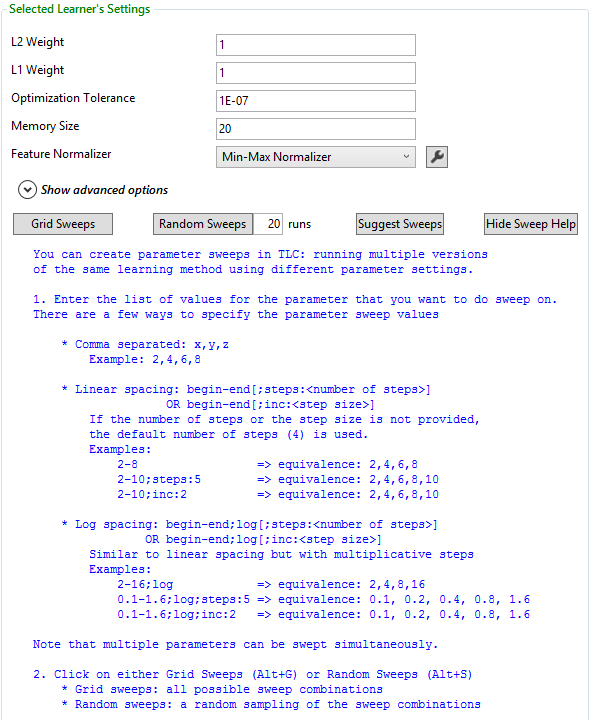
## Regularization

In machine learning algorithms, the user may want to control some aspects of the learned model (e.g., the L1 or L2 norms of the weight vector in linear classification algorithms, or the size of the tree or number of leaves in decision tree learning algorithms). Adding constraints to the algorithm regarding these aspects of the model that are independent of the training data is called *regularization*. For example, L1 regularization is done by subtracting the L1 weight of the weight vector from the loss expression that the learning algorithm is trying to minimize. The L1 norm is a good approximation to the L0 norm, which is the number of non-zero coordinates. Thus, L1 regularization is useful if the goal is to have a model that is as sparse as possible. L2 regularization prevents any single coordinate in the weight vector from growing too much in magnitude, so it is useful if the goal is to have a model with small overall weights.

Regularization is also used to avoid *overfitting*. Overfitting happens when the model is trained “too well”, such that it has very good success rate on the training set, but not necessarily on the test set. For example, suppose one wants to learn a classifier that categorizes documents. The features in this case include word counts (or frequencies) for the words that appear in the training documents. Suppose that word number image002.png appears only in training documents whose labels are positive. Therefore, during training, coordinate image002.png grows to the point where every test document that contains word image002.png is predicted positive, regardless of the other words in the document. Adding L1 and L2 regularizations in this case imposes a penalty on the magnitude of the parameter values. The algorithm tries to minimize the penalty, in a tradeoff with minimizing the loss.

## Using parameter sweeps

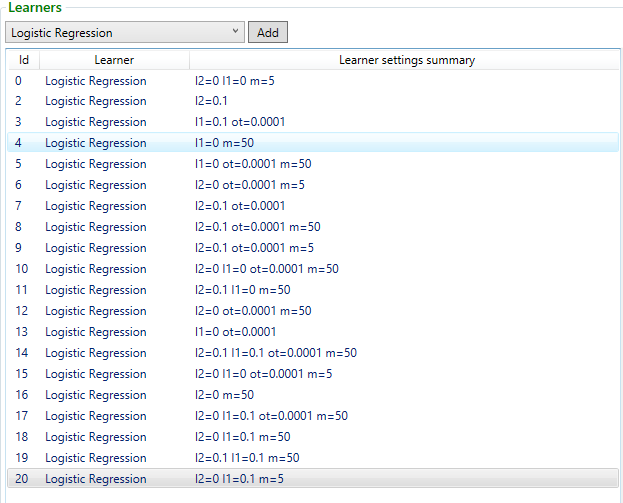
Many of the algorithms implemented in TLC have user defined parameters. These parameters can be used to run parameter sweeping. Parameter sweeping is a learning technique, where different models are trained by the same learning algorithm, using different parameters. Then the best prediction model can be chosen. In the TLC GUI, after choosing a learning algorithm, the parameter settings for the chosen algorithm appears in the right pane. You can also use parameter sweeps from the [command line](#ml_basics_using_parameter_sweeps_1621).



To run a parameter sweep, either manually specify the list of values for the parameter (or parameters) to do a sweep on, or click the “Suggest Sweeps” button. The list of values can be specified in one of the following ways:

* A comma-separated list, e.g. 2,4,6,8.
* Minimal and maximal value, separated by ‘-‘. This runs a sweep on four values in the range between the minimal and maximal values. E.g., entering 2-8 is equivalent to entering 2,4,6,8.
* Minimal and maximal value, separated by ‘-‘, followed by the number of linear steps, given by ;steps:<number of steps>. E.g., entering 2-10;steps:5 is equivalent to entering 2,4,6,8,10.
* Minimal and maximal value, separated by ‘-‘, followed by the linear step size, given by ;inc<step size>. E.g., entering 2-10;inc:2 is equivalent to entering 2,4,6,8,10.
* Adding the string ;log in any of the three last options, gives the same results but with a multiplicative step. E.g., entering 2-16;log is equivalent to entering 2,4,8,16. Entering 0.1-1.6;log;steps:5 or 1.6;log;inc:2 is equivalent to entering 0.1,0.2,0.4,0.8,1.6.

After entering the parameter values either manually or using the “Suggest Sweeps” button, either click “Grid Sweeps” to produce all the possible combinations of parameters, or choose a number of runs and click “Random Sweeps” for a random sample of the parameter combinations. Clicking one of these buttons produces a list of algorithms specifying the different values of the parameters in the “Learners” pane of the GUI:



### Using Parameter Sweeps from the Command Line

Similar parameter sweep functionality is available from the command line.  You specify that you are running sweeps and sweeper options.  The results are output to a file.

*On the Command Line:*

Use the command-line argument /c Sweep.

The following example runs 10 batches of 2 runs and defines 2 long parameters (ITER and BINS) to sweep on. It uses the Low Discrepancy Random Sweeper  with the Sobol sequence.

TL.exe /c Sweep /sbs=2 /snb=10 /out=c:\temp\  /sweeper=ldrand{/p=lp{name=ITER min=1 max=100 logbase+ inc=10} /p=lp{name=BINS min=2 max=10 inc=2} s+ } /pattern={..\UCI\adult.train /c=TrainTest /inst=TextInstances{header=+ sparse=- cat=1,3,5,6,7,8,9,13 sep=, label=14} /test=..\UCI\adult.test /cl=AveragedPerceptron{iter=$ITER$ norm=BinNormalizer{n=$BINS$}} }

**Sweeper Name and Options**

Specify the sweeper name and the sweeper options.  The following sweepers are available in recommended order:

* Low Discrepancy Random Sweeper (ldrand)
* Random Grid Sweeper (randomgrid)
* Uniform Grid Sweeper (uniformgrid)

*On the Command Line:*

* Use the command-line argument /sweeper= followed by the sweeper name
* The settings for the sweeper are passed by adding {/p= options} after the sweeper name
* Add the optional Sobol sequence for the low discrepancy random sweeper by adding s+ after the options

In the sweeper options, set the parameter type to sweep on, the parameter name, and either the min and max values of the possible values for discrete parameters.

|  |  |  |
| --- | --- | --- |
| **/p** | paramtype{options} | The parameter to sweep on. Can be long (lp), float (fp) value or discrete (dp).  For long or float value parameters      Required options are:           name - the variable name for the parameter           min - the minimum value           max - the maximum value      Optional entries are:           steps or inc - Choose to use evenly spaced steps or a set increment           log - use logarithmic spacing  For discrete value paramaters:       Required options are:           name - the variable name for the parameter           v - possible value for the parameters.  Use multiple times for each possible value. |
| **s+** |  | (optional) Use the Sobol sequence instead of the Niderreiter quasi-random sequence for Low Discrepancy Random Sweeper |
| **maxpoints** | integer | (optional) Set the maximum number of points for the Random Grid sweeper |

**Number of experiments**

Set the number of experiments per batch.

*On the Command Line:*

Use the command-line argument /sbs= followed by the number of experiments. Default is 5.

**Number of sets of experiments**

Set the number of experiment batches.   For example if you set number of batches to 5 and number of experiments to 2, you will have 5 sets of 2 experiments each.

*On the Command Line:*

Use the command-line argument /snb= followed by the number of batches.  Default is 5.

**Experiment Pattern**

Specifies the command line experiment to run - replacing the parameters to sweet on with variable names.

*On the Command Line:*

Use the command-line argument /pattern= followed by the experiment details.

**Output Folder**

The results will be saved to a file whose location is specified by the output folder.

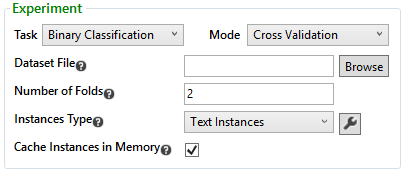
*On the Command Line:*

Use the command-line argument /out= followed by the folder path.

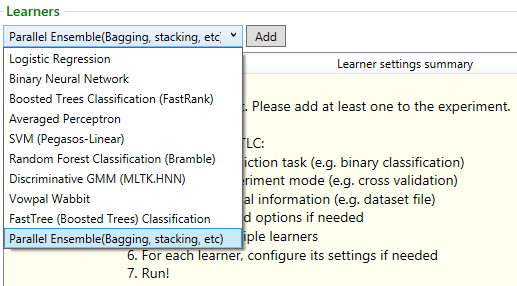
# TLC Learners

The TLC package implements five types of learning algorithms; binary classification algorithms, multi-class classification algorithms, regression algorithms, anomaly detection algorithms and ranking algorithms. This section contains an overview of each algorithm, and how to use them using TLC.

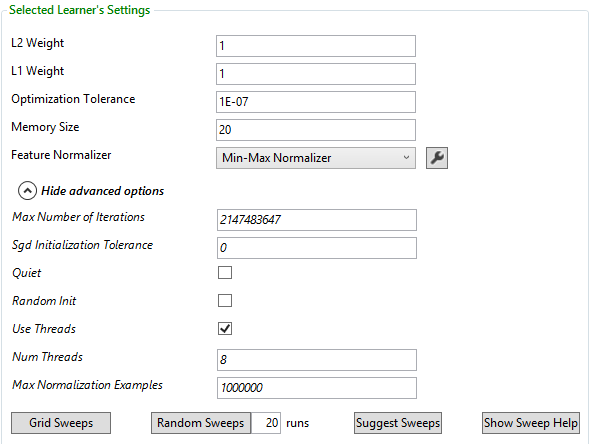
To train or use prediction models via TLC GUI, first select the task from the task drop list



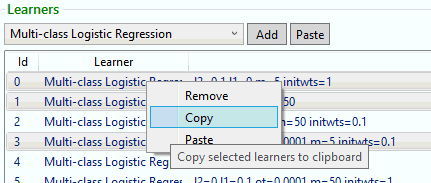
Then choose the algorithm from the learners drop list, and click the “Add” button next to it.



Each algorithm has its own user defined parameters. Once an algorithm is chosen from the list, the settings for it appear in the Selected Learner’s Settings in the right pane of the TLC GUI:



The learner with its settings can be copied by right clicking on the learner line and choosing "copy", and pasted either by right clicking again on the learners pane or by using the "paste" button:



This is useful for creating parameter sweeps, when trying to create a new learner where all the settings except for a few are the same.

*On the Command Line:*

* Use the command-line argument /cl followed by the algorithm name to pass it to the program.
* TL.exe /cl AlgorithmName {?} or TL.exe ? AlgorithmName displays a list of user defined parameters
* The parameters are passed to the program after the algorithm name in the *learner settings string*, in the format { parameter1=value1 parameter2=value2 … }

## Logistic regression

### Overview

Logistic Regression is a method in statistics used to predict the probability of occurrence of an event and can be used as a classification algorithm. The algorithm predicts the probability of occurrence of an event by fitting data to a logistical function. It is a regression technique that is naturally suited for data with binary labels. It assumes a logistic distribution on the data, where the probability that an example belongs to class 1 is image002.png. image004.png is the logistic distribution function, image006.pngis a image008.png dimensional vector containing the values of all the features of the instance, and image010.png are the unknown parameters of the logistic distribution. The algorithm tries to find the optimal values for image010.png by trying to maximize the log probability of the parameters given the input. The maximization is performed by using a technique called L-BFGS (see [Wikipedia](http://en.wikipedia.org/wiki/L-BFGS)).

TLC also implements a multi-class version of Logistic Regression, which has the same settings as the binary classification version.

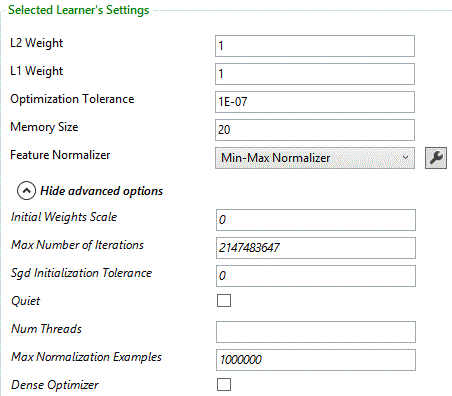
For more information, see the [Wikipedia entry for logistic regression](http://en.wikipedia.org/wiki/Logistic_regression) or the paper [Scalable Training of L1-Regularized Log-Linear Models](http://research.microsoft.com/apps/pubs/default.aspx?id=78900)by Andrew and Gao*.*

### Settings

*On the Command Line:*

* The command-line argument for Logistic Regression is /cl LR.
* The command-line argument for the multi-class version is /cl MultiClassLogisticRegression.

After selecting Logistic Regression from the learners drop list, the algorithm’s settings are displayed in the right pane of the TLC GUI.



#### L1 Weight, L2 Weight

This learner can use a linear combination of L1 and L2 regularizations. The default value for both coefficients is 1.

*On the Command Line:*

Add l1= or l2= followed by their values to the learner settings string.

#### Optimization Tolerance

This parameter sets the threshold for the optimizer convergence. In other words, if the improvement between iterations is less than the threshold, the algorithm stops and returns the current model. The default value of this parameter is 1E-07.

*On the Command Line:*

Add ot= followed by the threshold to the learner settings string.

#### Memory Size

The technique used for optimization here is L-BFGS, which uses only a limited amount of memory to compute the next step direction. This parameter indicates the number of past positions and gradients to store for the computation of the next step. The default value of this parameter is 20.

*On the Command Line:*

Add m= followed by the value to the learner settings string.

#### Feature Normalizer

By default, the features are normalized using the min-max normalizer. To specify no normalization, choose <None> from the Normalizer drop list. A different normalizer can also be chosen using this drop list.

*On the Command Line:*

* Add norm={} to the learner settings string to indicate no normalization
* Add norm=GaussianNormalizer or norm=BinNormalizer to the learner settings string to indicate normalization using a different normalizer. If a normalizer settings string is needed it follows the normalizer name, in a pair of curly brackets

#### Initial Weights Scale

The initial weights are initialized randomly. This parameter indicates the diameter of their initialization, i.e., if the diameter is specified to be d, the weights are uniformly distributed between -d/2 and d/2. The default value is 0, indicating all weights are initialized to 0.

*On the Command Line:*

* Add initwts= followed by its value to the learner settings string
* initwts=0 initializes all the weights to 0

#### Max Number of Iterations

Max number of iterations indicates the maximum number of optimization steps the algorithm should make. After this number of steps, the algorithm stops even if it hasn’t reached convergence.

*On the Command Line:*

Add maxiter= followed by the maximum number of iterations to the learner settings string.

#### Sgd Initialization Tolerance

If the value is changed to a number greater than 0, SGD is used to find the initial parameters, converging to the tolerance indicated.

*On the Command Line:*

Add sgd= followed by the value to the learner settings string.

#### Quiet

If this check box is checked, no output is produced during training.

*On the Command Line:*

Add q=+ to the learner settings string to indicate no output during training.

#### Use Threads, Num Threads

The “Use Threads” check box indicates whether or not to use threads, and Num thread is the number of threads (should be equal to the number of cores on the machine).

*On the Command Line:*

* Add t=- to the learner settings string to indicate no threads
* Add nt= followed by the number of threads to the learner settings string

#### Max Normalization Examples

The maximal number of instances that the normalizer processes before computing the normalization function. The default value is 1 million. This option is used to speed up the learning in cases where there is a large number of examples.

*On the Command Line:*

Add numNorm= followed by the value to the learner settings string.

#### Dense Optimizer

By default, this check box is unchecked, indicating that the logistic regression optimizer can use sparse or dense internal states as it finds appropriate.  Turning this on forces the internal optimizer to use a dense internal state, which may help alleviate load on the garbage collector on some varieties of larger problems.

*On the Command Line:*

Add do=+ to the learner settings string to use a dense internal optimizer state .

### Weighting examples

Logistic Regression supports weighting examples. To enable this option, weights should be specified in the input data file, and passed to the parser in the [Weight Column Idx](#input_data_format_text_input_for_5430) text box. Using weighted examples implies also that metrics are computed both weighted and unweighted. The latter being often useful in better weight impact understanding.

## Neural networks

### Overview

A neural network is a class of prediction models inspired by the human brain. A neural network can be thought of as a weighted directed graph. Each node in the graph is called a n*euron*. The neurons in the graph are arranged in layers, where neurons in one layer are connected by a weighted edge to neurons in the next layer (weights can be 0 or positive numbers). The first layer is called the *input layer*, and each neuron in the input layer corresponds to one of the features. The last layer of the function is called the *output layer* and in the case of binary neural networks it contains two output neurons, one for each class, whose values are the probabilities of belonging to each class. The remaining layers are called *hidden layers*. The values of the neurons in the hidden layers and in the output layer are set by calculating the weighted sum of the values of the neurons in the previous layer and applying an activation function to that weighted sum. A neural network model is defined by the structure of its graph (namely, the number of hidden layers and the number of neurons in each hidden layer), the choice of activation function, and the weights on the graph edges. The neural network algorithm tries to learn the optimal weights on the edges based on the training data. TLC provides a neural network specification language called Net# (see the [Net# user manual](#net_sharp_(cloudml_neural_networ_6785)), that can be used to define networks with complex structures, such as multiple hidden layers (a.k.a *deep learning* networks), full or filtered connections between layers, convolutional network connections and local response normalization layers.

TLC also implements a multi-class version of Neural Networks, and a regression version of Neural Networks. The settings for these versions are the same as the binary classification version, with two differences: The first difference is in the network architecture in the multi-class classification version consists of multiple neurons (one for every class) in the output layer whereas in the regression version it consists of a single neuron in the ouput layer, which outputs the predicted value of the function. The second difference is in the calibration settings, which are only needed for classification.

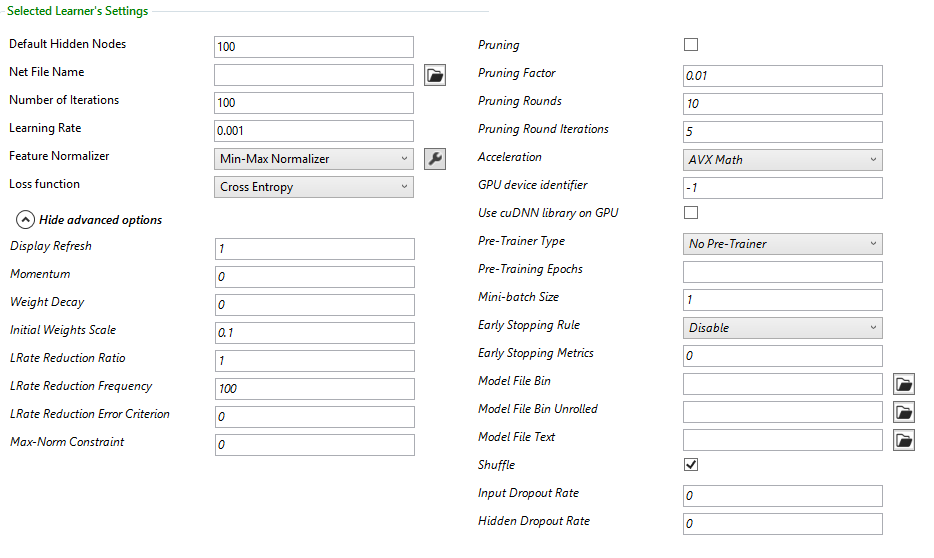
For more information, see the [Wikipedia entry for artificial neural network](http://en.wikipedia.org/wiki/Artificial_neural_network).

### Settings

*On the Command Line:*

* The command-line argument for Binary Neural Networks is /cl BinaryNeuralNetwork.
* The command-line argument for Multi-class Neural Networks is /cl MultiClassNeuralNetwork.
* The command-line argument for Binary Neural Networks is /cl RegressionNeuralNetwork.

After selecting Binary Neural Network from the learners drop list, the algorithm’s settings are displayed in the right pane of the TLC GUI.



#### Default hidden nodes

The default number of hidden nodes in the neural net. The default value is 100.

*On the Command Line:*

Add hidden= followed by the number of hidden nodes to the learner settings string.

#### Net File Name

This is a file that describes the architecture of the net being trained. Nets with arbitrary architectures are supported. Fully and sparsely connected nets can be created. The syntax of the net architecture is described in the [Net# user manual](#net_sharp_(cloudml_neural_networ_6785).  The default net structure is a fully connected directed graph, containing one neuron for each feature in the input layer, one neuron for each class in the output layer, and a hidden layer consisting of 100 neurons. This default structure is not intended for serious machine learning, but only for quick experimentation, and can be customized based on the structure of the output net structure.

In the multi-class case, if the net structure is not specified in Net File Name, the number of output nodes must be specified in the Default Output Nodes text box. This number must be specified as the number of classes.

*On the Command Line:*

* Add filename= followed by the name of the file containing the net architecture to the learner settings string
* For multi-class Neural Networks, if using the default net structure, add output= followed by the number of classes to the learner settings string

#### Number of iterations

The number of iterations on the full training set. The default value is 100.

*On the Command Line:*

Add iter= followed by the number of iterations to the learner settings string.

#### Learning Rate

Determines the size of the step taken in the direction of the gradient in each step of the learning process. The default learning rate is 0.001.

*On the Command Line:*

Add lr= followed by the learning rate to the learner settings string.

#### Feature Normalizer

By default, the features are normalized using the min-max normalizer. To specify no normalization, choose <None> from the Normalizer drop list. A different normalizer can also be chosen using this drop list.

*On the Command Line:*

* Add norm={} to the learner settings string to indicate no normalization
* Add norm=GaussianNormalizer or norm=BinNormalizer to the learner settings string to indicate normalization using a different normalizer. If a normalizer settings string is needed it follows the normalizer name, in a pair of curly brackets

#### Loss function

The default loss function used is cross entropy (log loss). The other loss available is squared loss.

*On the Command Line:*

Add loss=SquaredError to the learner settings string to indicate squared error loss.

#### **Display Refresh**

The frequency (in iterations) by which the console is updated with the latest information.

*On the Command Line:*

Add refresh= followed by the frequency (in iterations) to the learner settings string.

#### Momentum

The momentum term is used during training to weigh the weight updates of the previous iterations. By default its value is 0.

*On the Command Line:*

Add mom= followed by the value to the learner settings string.

#### Weight Decay

After each weight update, the weights in the network are scaled by (1 - learning rate \* weight decay). The default value is 0.

*On the Command Line:*

Add wd= followed by the value to the learner settings string.

#### Initial Weights Scale

The initial weights of the network edges are initialized randomly. This parameter indicates the diameter of their initialization, i.e., if the diameter is specified to be d, the weights are uniformly distributed between -d/2 and d/2. The default value is 0.1.

*On the Command Line:*

* Add initwts= followed by its value to the learner settings string
* initwts=0 initializes all the weights to 0. This is typically not recommended for Neural networks

#### **Lrate Reduction Ratio, Lrate Reduction Frequency, Lrate Reduction Error Criterion**

LRate Reduction Ratio is the ratio by which the learning rate is reduced during training. Reducing the learning rate can avoid local minima. A value of 1.0 means no reduction. A value of 0.9 means the learning rate is reduced to 90% of its current value. The reduction can be either triggered periodically (every fixed number of iterations), or when a certain error criterion is met.

To trigger rate reduction periodically, specify the frequency, in number of iterations in “LRate Reduction Frequency”. E.g., if 10 is specified, the learning rate is reduced once every 10 iterations.

To trigger rate reduction based on an error criterion, specify a number in “LRate Reduction Error Criterion”. A value of 0, means that only if the loss increases between iterations the learning rate is reduced. A fractional value of image008.png which is greater than 0, means that the learning rate is reduced if the loss decreases by less than an image008.png fraction of its previous value.

*On the Command Line:*

* Add lred= followed by the reduction ratio to the learner settings string
* Add lredfreq= followed by the frequency to the learner settings string
* Add lrederror= followed by the value of the error criterion to the learner setting string

#### Max-Norm Constraints

This option allows to constrain the norm of the incoming weight vector at each hidden unit to be upper bounded by the constant. Max-norm can be very important in maxout neural networks as well as in cases where training produces unbounded weights (which  leads to “Found NaNs or infinity in network parameters” message during the training).

*On the Command Line:*

Add maxnorm=  followed by the constraint as a floating point value to the learner settings string.

#### Early Stopping Rule

This feature can decide the number of iterations based on the result of validation tests so that the training doesn't fall into overfitting. For the detail of these rules, please see this paper: *Lodwich, Aleksander, Yves Rangoni, and Thomas Breuel, "*[*Evaluation of robustness and performance of Early Stopping Rules with Multi Layer Perceptrons*](http://iupr1.cs.uni-kl.de/~shared/publications/2009-lodwich-ijcnn-robust-performance-stop-rule.pdf)*," Neural Networks, 2009.*

To enable early stopping, select one of the early stopping rules from the drop-down in the GUI and add the [validation dataset](#learning_modes_train_mode_htm_va_3838).  Possible parameter values for the rules are threshold (**TH**) and/or window size (**W**).

|  |  |  |  |
| --- | --- | --- | --- |
| Early stopping rule | Short Name | Params | When it stops |
| Tolerant | **TR** | **TH** | Stops when "validationScore" < "bestScore" - **TH** |
| Loss of Generality | **GL** | **TH** | Stops when "validationScore" < "bestScore" \* (1 - **TH**) |
| Low Progress | **LP** | **TH, W** | Stops when "recentAvg" >= (1 -**TH**) \* "recentBest" |
| Generality to Progress rate | **PQ** | **TH, W** | Stops when "validationScore" \* "recentBest" <= (1 - **TH**) \* "bestScore" \* "recentAvg" |
| Consecutive Loss in Generality | **UP** | **W** | Stops when the score degraded **W** times in a row. |

Where:

* validationScore: the score of the latest validation test.
* bestScore: the best ever validation score.
* recentBest: the best training score in the past **W** steps.
* recentAvg: the average training score in the past **W** steps.

*On the Command Line:*

* Add esr= followed by the early stopping rule and paramaters to the learner settings string
* Add esmt= followed by the number that indicates the early stopping metric to the learner settings string.

You can choose the early stopping metric:

* For FastTree classification, error rate (0) is always used.
* For FastTree regression,  L1 (1) is the default or select L2 (2)
* For FastTree ranking, NDCG@1 (1) is the default or select NDCG@3 (3)
* For Neural Nets mean error of NN outputs (0) is always used

#### **Acceleration**

The hardware acceleration option, which could be CLR, SSE, AVX, or GPU. AVX is the default option, falling back to SSE if it is not supported, or CLR if SSE is also not supported.

Currently the only GPU devices supported by TLC are CUDA-capable devices of [Compute Capability 2.0](http://developer.nvidia.com/cuda-gpus) and higher. Additionally, please make sure that:

* The CUDA 5.5 binaries have been copied from [\\cloudmltlc\TLC\NonMicrosoftLibraries\CUDA\5.5](file:/cloudmltlc/TLC/NonMicrosoftLibraries/CUDA/5.5) to the TLC binaries folder.
* The latest graphics drivers from [NVIDIA website](http://www.nvidia.com/page/drivers.html) are installed.

The July release introduces a new acceleration option: Intel MKL. This option allows to significantly speed up the training of some NN configurations on CPU. Performance improvement is especially noticeable when used with fully connected bundles and mini-batch size of 8 or larger.

Note that Intel MKL will most likely be slower than existing SSE/AVX acceleration options when training with mini-batch of size 1 so  it is always recommended to compare performance using different acceleration options.

*On the Command Line:*

Add accel= followed by CLR, SSE or GPU to the learner settings string.

#### GPU device identifier

This option is used only when the [hardware acceleration platform](#tlc_learners_neural_networks_htm_1280) is GPU. In case the system has multiple GPUs, it is possible to specify which one to use, by specifying a GPU device id, which is a number between 0 and the number of GPUs minus 1. The default value is -1, in which case the GPU is chosen automatically by the system.

*On the Command Line:*

Add gpuid= followed by the id to the learner settings string.

#### Use cuDNN library on GPU

Checking this option switches from TLC convolution implementation to the one from NVIDIA cuDNN library (<https://developer.nvidia.com/cuDNN> ). The main advantage of cuDNN library is very low GPU memory utilization so larger nets can be trained on GPUs. Using this option may also speed up the training by 10-15% in some cases.

*On the Command Line:*

Add cudnn+ to the learner settings string.

#### Pre-Trainer Type, Pre Training Epochs

Some nets, especially nets with more than one hidden layer, may require pre-training in order to converge. Currently one form of pre-training is supported (layer-wise greedy pre-trainer). Other pre-training techniques will be added in the next releases.

Pre Training Epoch determines the number of iterations on the full training set in the pre-training phase. If it is not set, the default is the [number of iterations](#tlc_learners_neural_networks_htm_9983).

*On the Command Line:*

* Add pretrain=Greedy to the learner settings string to indicate pre-training is required
* Add prepoch= followed by the number of iterations to the learner settings string

#### Mini-batch size

The Neural Networks algorithm uses Stochastic Gradient Descent to update the edge weights in each iteration. By default, each iteration uses one instance from the training set to update the weights. When AVX or GPU are available, the mini-batches option is supported. When mini-batches are enabled, each iteration uses a small batch of instances to update the weights. Using mini-batches improves training time by doing matrix-matrix computations instead of matrix-vector computations. Matrix-matrix operations are easily parallelized and extremely efficient on AVX or GPU. The downside of using mini-batches is slower convergence, but in medium/large datasets, training time with mini-batches is smaller, which allows experimentation with various net configurations. For extremely large datasets (like ImageNet), it is necessary to use mini-batches to get a reasonable running time.

The size of the mini-batch is determined by the value in the Mini-batch size text box. The default size of a mini-batch is 1 (which is equivalent to regular SGD), and the recommended size is between 32 and 256.

*On the Command Line:*

Add mbsize= followed by the size to the learner settings string.

#### Early Stopping Rule

This feature can decide the number of iterations based on the result of validation tests so that the training doesn't fall into overfitting. For the detail of these rules, please see this paper: *Lodwich, Aleksander, Yves Rangoni, and Thomas Breuel, "*[*Evaluation of robustness and performance of Early Stopping Rules with Multi Layer Perceptrons*](http://iupr1.cs.uni-kl.de/~shared/publications/2009-lodwich-ijcnn-robust-performance-stop-rule.pdf)*," Neural Networks, 2009.*

To enable early stopping, select one of the early stopping rules from the drop-down in the GUI and add the [validation dataset](#learning_modes_train_mode_htm_va_3838).  Possible parameter values for the rules are threshold (**TH**) and/or window size (**W**).

|  |  |  |  |
| --- | --- | --- | --- |
| Early stopping rule | Short Name | Params | When it stops |
| Tolerant | **TR** | **TH** | Stops when "validationScore" < "bestScore" - **TH** |
| Loss of Generality | **GL** | **TH** | Stops when "validationScore" < "bestScore" \* (1 - **TH**) |
| Low Progress | **LP** | **TH, W** | Stops when "recentAvg" >= (1 -**TH**) \* "recentBest" |
| Generality to Progress rate | **PQ** | **TH, W** | Stops when "validationScore" \* "recentBest" <= (1 - **TH**) \* "bestScore" \* "recentAvg" |
| Consecutive Loss in Generality | **UP** | **W** | Stops when the score degraded **W** times in a row. |

Where:

* validationScore: the score of the latest validation test.
* bestScore: the best ever validation score.
* recentBest: the best training score in the past **W** steps.
* recentAvg: the average training score in the past **W** steps.

*On the Command Line:*

* Add esr= followed by the early stopping rule and paramaters to the learner settings string
* Add esmt= followed by the number that indicates the early stopping metric to the learner settings string.

You can choose the early stopping metric:

* For FastTree classification, error rate (0) is always used.
* For FastTree regression,  L1 (1) is the default or select L2 (2)
* For FastTree ranking, NDCG@1 (1) is the default or select NDCG@3 (3)
* For Neural Nets mean error of NN outputs (0) is always used

#### Model File Bin, Model File Bin Unrolled, Model File Text

TLC has a new native predictor library, that can be used to deploy a neural network model in unmanaged environment (such as mobile devices). Specify a file name in the "Model File Bin Unrolled" text box, and then use Predictor::Load to construct the predictor object from the model file. Call predictor->Predict(const float\* features) to make predictions.

The model is saved without any convolutional structures that it may contain, and the edge weights are simply represented as a sparse matrix. To save the model including its convolutional structures, specify a file name in the "Model File Bin" (this format cannot be loaded by the unmanaged predictor).

A text file name can also be specified in the "Model File Text" text box, which saves the model in [text format](#output_model_formats_text_format_1011).

*On the Command Line:*

* Add mbu= followed by the file name to the learner settings string to save without convolutional structures
* Add mb= followed by the file name to the learner settings string to save with convolutional structures
* Add mt= followed by the file name to the learner settings string to save as text

#### Shuffle

This check box is checked by default, and indicates that the instances are shuffled before each training iteration.

*On the Command Line:*

Add shuf=- to the learner settings string to indicate no shuffling.

#### Input Droput Rate, Hidden Dropout Rate

When the training set is small, the Neural Networks algorithm typically performs poorly on the test set, due to overfitting. This overﬁtting can be reduced by randomly “dropping” (setting their activation level to 0) a fraction of the neurons in iteration. The “dropout” technique has been shown to improve performance on many benchmark speech and object recognition problems (see the paper [Improving neural networks by preventing co-adaptation of feature detectors](http://arxiv.org/abs/1207.0580) by Hinton et al.).

TLC supports two kinds of dropouts: Input dropout and hidden dropout. Both dropout rates are 0 by default. For each instance, every neuron in the input layer gets dropped independently with probability equal to Input Dropout Rate and every neuron in the hidden layers gets dropped independently with probability equal to Hidden Dropout Rate. Hidden Dropout Rate only applies to fully connected or filtered hidden layers, in other kinds of layers there is no dropout (see the [Net# user manual](#net_sharp_(cloudml_neural_networ_6785) for details on the different kinds of hidden layers). Recommended values for Input dropout rate are between 0 and 0.2, and the recommended value for hidden dropout rate is 0.5.

*On the Command Line:*

* Add idrop= followed by the input dropout rate to the learner settings string
* Add hdrop= followed by the hidden dropout rate to the learner settings string

## Boosted trees

The TLC FastRank wrapper has retired, and [FastTree](#tlc_learners_fasttree_htm) is replacing it. All the command line names for FastRank (FR, FastRankRegression, FastRankRanking etc.) will run FastTree. If the FastRank wrapper must be used, the command line /cl OldFastRankBinaryClassification,for binary classification, /cl OldFastRankRanking for ranking or /cl OldFastRankRegression for regression.

## Averaged perceptron

### Overview

Perceptron is a classification algorithm that makes its predictions based on a linear function. I.e., for an instance with feature values image002.png, the prediction is given by the sign of image004.png, where image006.png are the weights computed by the algorithm.

Perceptron is an online algorithm, i.e., it processes the instances in the training set one at a time. The weights are initialized to be 0, or some random values. Then, for each example in the training set, the value of image004.png is computed. If this value has the same sign as the label of the current example, the weights remain the same. If they have opposite signs, the weights vector is updated by either subtracting or adding (if the label is negative or positive, respectively) the feature vector of the current example, multiplied by a factor image008.png, called the learning rate. In a generalization of this algorithm, the weights are updated by adding the feature vector multiplied by the learning rate, and by the gradient of some loss function (in the specific case described above, the loss is hinge-loss, whose gradient is 1 when it is non-zero).

In Averaged Perceptron (AKA *voted-perceptron*), the weight vectors are stored, together with a weight that counts the number of iterations it survived (this is equivalent to storing the weight vector after every iteration, regardless of whether it was updated or not). The prediction is then calculated by taking the weighted average of all the sums image004.png for the different weight vectors.

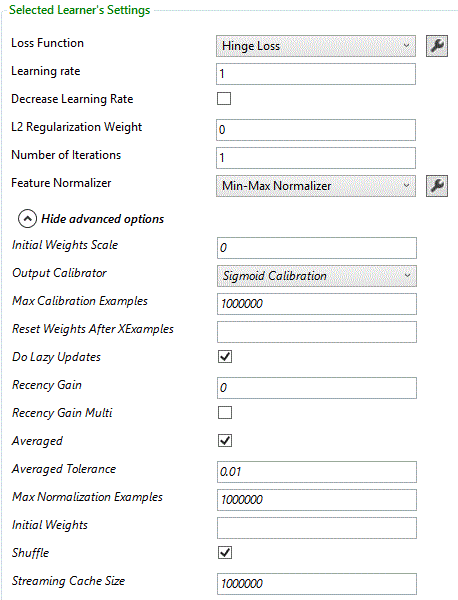
For more information, see the [Wikipedia entry for Perceptron](http://en.wikipedia.org/wiki/Perceptron), or the paper [Large Margin Classification Using the Perceptron Algorithm](http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.48.8200) by Y. Freund and R.E. Schapire. Another good description can be found in the paper [Discriminative Training Methods for Hidden Markov Models:Theory and Experiments with Perceptron Algorithms](http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.18.6725) by M. Collins.

### Settings

*On the Command Line:*

The command-line argument for Averaged Perceptron is /cl AP.

After selecting Averaged Perceptron from the learners drop list, the algorithm’s settings are displayed in the right pane of the TLC GUI.



#### Learning Rate

Determines the size of the step taken in the direction of the gradient in each step of the learning process. The default learning rate is 1.

*On the Command Line:*

Add lr= followed by the learning rate to the learner settings string.

#### Decrease Learning Rate

A flag indicating whether to reduce the learning rate as iterations progress. By default, the learning rate is not decreased. If the flag is turned on, then in iteration number image002.png the learning rate is reduced by a factor of image006.png. To indicate learning rate reductions check the “Decrease Learning Rate” check box.

*On the Command Line:*

Add decreaselr=+ (to indicate learning rate reductions) or decreaselr=- (to indicate no learning rate reductions) to the learner settings string.

**Tip:** If this check box is checked, it may improve results to uncheck the [Averaged](#tlc_learners_averaged_perceptron_6205) check box. This is because decreasing the learning rate and averaging have a similar “stabilizing” outcome (resulting in later iterations having less and less effect). One advantage that decreasing the learning rate has over averaging is that decreasing the learning rate guarantees convergence, while averaging doesn’t.

#### Loss function

The default loss function is hinge-loss. The other losses available are Poisson loss, exponential loss and squared loss.

*On the Command Line:*

Add loss= followed by the name of the loss to the learner settings string. Available losses are SquaredError, PoissonLoss and ExpLoss.

#### L2 Regularization Weight

The weights can be L2 regularized. By default, the weight of the regularization is 0.

*On the Command Line:*

Add reg= followed by the regularization coefficient to the learner settings string.

#### Number of iterations

The number of iterations on the full training set. The default value is 1.

*On the Command Line:*

Add iter= followed by the number of iterations to the learner settings string.

#### Feature Normalizer

By default, the features are normalized using the min-max normalizer. To specify no normalization, choose <None> from the Normalizer drop list. A different normalizer can also be chosen using this drop list.

*On the Command Line:*

* Add norm={} to the learner settings string to indicate no normalization
* Add norm=GaussianNormalizer or norm=BinNormalizer to the learner settings string to indicate normalization using a different normalizer. If a normalizer settings string is needed it follows the normalizer name, in a pair of curly brackets

#### Initial Weights Scale

The initial weights are initialized randomly. This parameter indicates the diameter of their initialization, i.e., if the diameter is specified to be d, the weights are uniformly distributed between -d/2 and d/2. The default value is 0, indicating all weights are initialized to 0.

*On the Command Line:*

* Add initwts= followed by its value to the learner settings string
* initwts=0 initializes all the weights to 0

#### Output Calibrator

Some classification algorithms produce raw scores that are unbounded. In order to convert these outputs into a binary prediction (or the predicted probability of belonging to one of the classes) the output must be calibrated. By default [sigmoid calibration](#introduction_classification_outp_9719). The other two options in the Calibrator drop list are [PAV calibration](#introduction_classification_outp_1316) and [Naïve calibrator](#introduction_classification_outp_6402).

*On the Command Line:*

* Add cali=PAVCalibration or cali=NaiveCalibration to the learner settings string to indicate calibration using a different calibrator
* Add cali={}to indicate no calibration

#### Max Calibration Examples

The maximal number of instances that are processed before computing the calibration function. The default value is 5 million. This option is used to speed up the calibration process in cases where there is a large number of examples.

*On the Command Line:*

Add numCali= followed by the value to the learner settings string.

#### Reset Weights After XExamples

Indicates the number of examples after which the weight vector is reset to the current weighted average.

*On the Command Line:*

Add numreset= followed by the value to the learner settings string.

#### Do Lazy Updates

Indicates that instead of updating averaged weights on every example, only update them when the loss is non-zero.

*On the Command Line:*

Add lazy=- to the learner settings string to indicate updating the weights on every example.

#### Recency Gain, Recency Gain Multi

If the value of the Recency Gain parameter is positive, it indicates that bigger weights should be given to more recent updates of the weight vector. In this case, the Recency Gain Multi check box should be checked if the recency gain should be multiplicative, and unchecked if it should be additive.

*On the Command Line:*

* Add rg= followed by the value of the recency gain to the learner settings string
* Add rgm=+ to the learner settings string to indicate multiplicative gain (default is additive)

#### Averaged

Unchecking this check box turns off averaging of the weight vectors of the different iterations.

*On the Command Line:*

Add avg=- to the learner settings string to turn off averaging.

#### Averaged Tolerance

A tolerance between 0 and 1, indicating the degree of tolerated inexactness in the averaging. Higher values will lead to faster execution of learning. This only has an effect if “averaged” is turned on.

*On the Command Line:*

Add avgtol= followed by the value to the learner settings string.

#### Max Normalization Examples

The maximal number of instances that the normalizer processes before computing the normalization function. The default value is 1 million. This option is used to speed up the learning in cases where there is a large number of examples.

*On the Command Line:*

Add numNorm= followed by the value to the learner settings string.

#### Initial Weights

A string containing the initial weights of the weight vector to be used by the algorithm. It is a comma-separated list of the weights.

*On the Command Line:*

Add initweights= followed by the initial weights string to the learner settings string.

**Tip:** This option can be used to output feature values. To do this, pass “0,0,0,…,1,0,…” as the initial weights string (with 1 in the location of the feature to output and 0 everywhere else), uncheck the “Calibrate Output” check box, and change the number of iterations to 0.

#### Shuffle

This check box is checked by default, and indicates that the instances are shuffled before each training iteration.

*On the Command Line:*

Add shuf=- to the learner settings string to indicate no shuffling.

#### Streaming Cache Size

This parameter indicates the size of the cache when trained in Scope. The default value is 1000000.

*On the Command Line:*

Add cache= followed by its value to the learner settings string to indicate the size of the cache when trained in Scope.

## SVM

### Overview

The idea behind support vector machines, is to map the instances into a high dimensional space in which instances of the two classes are linearly separable, i.e., there exists a hyperplane such  that all the positive examples are on one side of it, and all the negative examples are on the other. After this mapping, quadratic programming is used to find the separating hyperplane that maximizes the *margin*, i.e., the minimal distance between it and the instances.

In Linear SVM, the high dimensional space is the feature space. The Pegasos algorithm alternates between stochastic gradient descent steps and projection steps, to find the weight vector image002.png, which minimizes the function

image004.png

where image006.png is the number of training instances, and the loss function is hinge-loss:

image008.png

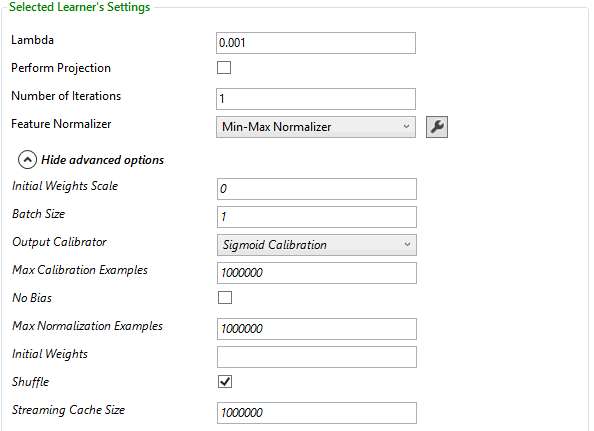
(image010.png is the feature vector of example image012.png, and image014.png is its label in image016.png. image018.png is called the regularizer constant. The prediction of the classifier for an instance image020.png is image022.png).

For more information, see the [Wikipedia entry for Support Vector Machines](http://en.wikipedia.org/wiki/Support_vector_machine), or the paper [Pegasos: Primal Estimated sub-GrAdient SOlver for SVM](http://www.cs.huji.ac.il/~shais/papers/ShalevSiSrCo10.pdf) by S. Shalev-Shwartz et al.

### Settings

*On the Command Line:*

The command-line argument for SVM is /cl LinearSVM.

After selecting SVM (Pegasos-Linear) from the learners drop list, the algorithm’s settings are displayed in the right pane of the TLC GUI.  


#### Lambda

The regularizer constant. It is also inversely proportional to the learning rate. The default value is 0.001.

*On the Command Line:*

Add lr= followed by the value to the learner settings string.

#### Perform Projection

This parameter indicates whether projection should be performed. By default, it is not performed, as it slows down the algorithm.

*On the Command Line:*

Add project=+ to the learner settings string to indicate projection should be performed.

#### Number of iterations

The number of iterations on the full training set. The default value is 1.

*On the Command Line:*

Add iter= followed by the number of iterations to the learner settings string.

#### Feature Normalizer

By default, the features are normalized using the min-max normalizer. To specify no normalization, choose <None> from the Normalizer drop list. A different normalizer can also be chosen using this drop list.

*On the Command Line:*

* Add norm={} to the learner settings string to indicate no normalization
* Add norm=GaussianNormalizer or norm=BinNormalizer to the learner settings string to indicate normalization using a different normalizer. If a normalizer settings string is needed it follows the normalizer name, in a pair of curly brackets

#### Initial Weights Scale

The initial weights are initialized randomly. This parameter indicates the diameter of their initialization, i.e., if the diameter is specified to be d, the weights are uniformly distributed between -d/2 and d/2. The default value is 0, indicating all weights are initialized to 0.

*On the Command Line:*

* Add initwts= followed by its value to the learner settings string
* initwts=0 initializes all the weights to 0

#### Batch size

The Batch Size parameter indicates the size of the sample to be used in each step of training. The default value is 1.

*On the Command Line:*

Add batch= followed by the sample size to the learner settings string.

#### Output Calibrator

Some classification algorithms produce raw scores that are unbounded. In order to convert these outputs into a binary prediction (or the predicted probability of belonging to one of the classes) the output must be calibrated. By default [sigmoid calibration](#introduction_classification_outp_9719). The other two options in the Calibrator drop list are [PAV calibration](#introduction_classification_outp_1316) and [Naïve calibrator](#introduction_classification_outp_6402).

*On the Command Line:*

* Add cali=PAVCalibration or cali=NaiveCalibration to the learner settings string to indicate calibration using a different calibrator
* Add cali={}to indicate no calibration

#### Max Calibration Examples

The maximal number of instances that are processed before computing the calibration function. The default value is 5 million. This option is used to speed up the calibration process in cases where there is a large number of examples.

*On the Command Line:*

Add numCali= followed by the value to the learner settings string.

#### No Bias

By default, the TLC implementation of SVM includes a bias term in the weights vector. In some cases, this term may increase the accuracy of the classifier. When a bias term (denoted image029.png) is present, the prediction for an instance image020.png becomes image031.png, and the loss is accordingly defined as

image033.png

To indicate image035.png check the No Bias check box.

*On the Command Line:*

Add n=+ to the learner settings string to indicate b=0.

#### Max Normalization Examples

The maximal number of instances that the normalizer processes before computing the normalization function. The default value is 1 million. This option is used to speed up the learning in cases where there is a large number of examples.

*On the Command Line:*

Add numNorm= followed by the value to the learner settings string.

#### Initial Weights

A string containing the initial weights of the weight vector to be used by the algorithm. It is a comma-separated list of the weights.

*On the Command Line:*

Add initweights= followed by the initial weights string to the learner settings string.

**Tip:** This option can be used to output feature values. To do this, pass “0,0,0,…,1,0,…” as the initial weights string (with 1 in the location of the feature to output and 0 everywhere else), uncheck the “Calibrate Output” check box, and change the number of iterations to 0.

#### Shuffle

This check box is checked by default, and indicates that the instances are shuffled before each training iteration.

*On the Command Line:*

Add shuf=- to the learner settings string to indicate no shuffling.

#### Streaming Cache Size

This parameter indicates the size of the cache when trained in Scope. The default value is 1000000.

*On the Command Line:*

Add cache= followed by its value to the learner settings string to indicate the size of the cache when trained in Scope.

## Random forests (Bramble)

Random forests (Bramble) is a framework for classification, regression, density estimation, manifold learning and semi-supervised learning developed at Microsoft Research Cambridge. TLC provides a wrapper only for the binary classification algorithm. For details and tutorials on how to use Bramble see [the Bramble website](http://research.microsoft.com/apps/pubs/default.aspx?id=158806), and [introduction to Bramble](file://cloudmltlc/tlc/doc/bramble). For general information about random forests, see [Wikipedia](http://en.wikipedia.org/wiki/Random_forest).

*On the Command Line:*

The command-line argument for Bramble is /cl RFC.

MLTK stands for Machine Learning Tool Kit. It is a collection of various machine learning algorithms and tools. Initially it was implemented by Mikhail Parakhin in Windows and now it is owned and developed by Domains Relevance Team. For information on how to use this algorithm see the document located [here](file://cloudmltlc/tlc/doc/mltk).

TLC also implements a multi-class version of Discriminative GMM, which has the same settings as the binary classification version.

*On the Command Line:*

* The command-line argument for GMM (MLTK.HNN) is /cl HNN.
* The command-line argument for the multi-class version is /cl HNNMultiClassClassifier.

## Vowpal Wabbit

Vowpal Wabbit is a machine learning library developed originally at Yahoo! Research, and currently at Microsoft Research started and lead by John Langford. For details and tutorials on how to use Vowpal Wabbit see the [Vowpal Wabbit website](http://hunch.net/~vw/) or [Wikipedia](http://en.wikipedia.org/wiki/Vowpal_Wabbit).

**Note:** The TLC GUI runs Vowpal Wabbit without overhead. To use, download libvw.dll from <http://toolbox/vw> and place it in the directory containing the rest of the TLC binaries.

*On the Command Line:*

The command-line argument for Vowpal Wabbit is /cl VW.

## FastTree

### Overview

FastTree is a TLC implementation of FastRank. astRank is an efficient implementation of the MART gradient boosting algorithm. MART learns an ensemble of regression trees, which is a decision tree with scalar values in its leaves.

A decision tree (or a regression tree) is a binary tree-like flow chart, where at every interior node, one decides which of the two child nodes to continue to, based on the value of one of the features of the input. In each leaf node, a value is returned. In the interior nodes, the decision is based on the test “image002.png?”, where image004.png is the value of the feature in the input sample and  image006.png is one of the possible values of this feature. The functions that can be produced by a regression tree are all the piece-wise constant functions.

The ensemble of trees is produced by computing, in each step, a regression tree that approximates the gradient of the loss function, and adding it to the previous tree with coefficients that minimize the loss of the new tree.

The output of the ensemble produced by MART on a given instance is the sum of the tree outputs. In case of a binary classification problem, the output is converted to probability by using some form of calibration. In case of a regression problem, the output is the predicted value of the function. In case of a ranking problem, the instances are ordered by the output value of the ensemble.

TLC also implements a regression version of FastRank, and a ranking version of FastRank. In the ranking case, the instances should be ordered by the output of the tree ensemble. The only difference in the settings of these versions is in the calibration settings, which are needed only for classification.

[Here](http://research.microsoft.com/en-us/um/people/cburges/talks/rankingBriefly.pptx) is a presentation with audio that explains LambdaMART in a nutshell (5 minutes). Also, see the paper [From RankNet to LambdaRank to LambdaMART: An Overview](http://research.microsoft.com/apps/pubs/default.aspx?id=132652) by C.J.C. Burges.

For more information, see the [Wikipedia entry for boosted trees, section about gradient tree boosting](http://en.wikipedia.org/wiki/Boosted_trees#Gradient_tree_boosting) or the paper [Greedy Function Approximation: A Gradient Boosting Machine](http://projecteuclid.org/DPubS?service=UI&version=1.0&verb=Display&handle=euclid.aos/1013203451) by J.H. Friedman.

### Settings

*On the Command Line:*

* The command-line argument for FastRank is /cl FR.
* The command-line argument for the regression version is /cl FastRankRegression.
* The command-line argument for the ranking version is /cl FastRankRanking.

After selecting FastRank from the learners drop list, the algorithm’s settings are displayed in the right pane of the TLC GUI.

#### Num Trees

Indicates the total number of trees constructed. The default value is 100. To change this value, enter the new value in the “Num Trees” box.

*On the Command Line:*

Add iter= followed by the number of trees to the learner settings string.

#### Num Leaves

Indicates the maximum number of leaves per tree. The default number is 20. To change this value, enter the new value in the “Num Leaves” box.

*On the Command Line:*

Add nl= followed by the number of leaves to the learner settings string.

#### Min Documents In Leaf

Indicates the minimum number of training instances required to form a leaf. The default number is 10. The number of instances is counted even if instances are weighted (not the total weight). To change this value, enter the new value in the “Min Instances In Leaf” box.

*On the Command Line:*

Add mil= followed by the value to the learner settings string.

#### Learning Rate

Determines the size of the step taken in the direction of the gradient in each step of the learning process. The default learning rate of Neural Networks is 0.001, the default learning rate of FastRank is 0.2, the default learning rate of Averaged Perceptron is 1 and the default learning rate of SGD is 0.1.

*On the Command Line:*

Add lr= followed by the learning rate to the learner settings string.

#### Advanced options

FastRank has a huge number of parameters that can be defined. In most cases, none of them need to be changed. Here is an overview of some of the important advanced parameters.

#### Output Calibrator

Some classification algorithms produce raw scores that are unbounded. In order to convert these outputs into a binary prediction (or the predicted probability of belonging to one of the classes) the output must be calibrated. By default [sigmoid calibration](#introduction_classification_outp_9719). The other two options in the Calibrator drop list are [PAV calibration](#introduction_classification_outp_1316) and [Naïve calibrator](#introduction_classification_outp_6402).

*On the Command Line:*

* Add cali=PAVCalibration or cali=NaiveCalibration to the learner settings string to indicate calibration using a different calibrator
* Add cali={}to indicate no calibration

#### Write Last Ensemble

Write the last ensemble instead of the one determined by [early stopping](#tlc_learners_fasttree_htm_early__8685).

*On the Command Line:*

Add hl=+ to the learner settings string to indicate writing the last ensemble.

#### Rng Seed

The seed used for the random number generator (needed for repro purposes). Randomness is used for sampling instances to create bins for the feature values.

*On the Command Line:*

Add r1= followed by the value to the learner settings string.

#### Test Frequency

The frequency with which the algorithm tests its current model on the validation set. It is used only of a validation dataset is supplied.

*On the Command Line:*

Add tf= followed by the frequency to the learner settings string.

#### Entropy Coefficient

The entropy (regularization) coefficient between 0 and 1. Determines how balanced the trees should be in the sense of number of instances that fall in each side. The larger this coefficient, the more balanced the splits are.

*On the Command Line:*

Add e= followed by the entropy coefficient to the learner settings string.

#### Histogram Pool Size

The number of histograms in the pool. This parameter only affects the speed. -1 indicates caching the whole history.

*On the Command Line:*

Add histpool= followed by the number of histograms to the learner settings string.

#### Max Bins

Determines the maximum number of bins to create for each feature. If the feature has less values than the number indicated, each value is in its own bin. If there are more values, the algorithm creates MaxBins bins.

*On the Command Line:*

Add mb= followed by the number of bins to the learner settings string.

#### Feature Fraction

The fraction of features (chosen randomly) to use for each tree. The default value is 1.

*On the Command Line:*

Add ff= followed by the value to the learner settings string.

#### Feature First Use Penalty

This is a form of regularization that incurs a penalty for using a new feature when creating the tree. Increase this value to create trees that don’t use many features.

*On the Command Line:*

Add ffup= followed by the value to the learner settings string.

#### Feature Reuse Penalty

This is a form of regularization that incurs a penalty for using a feature that has already been used in the same tree.

*On the Command Line:*

Add frup= followed by the value to the learner settings string.

#### Split Fraction

The fraction of features (chosen randomly) to use on each split. The default value is 1.

*On the Command Line:*

Add sf= followed by the value to the learner settings string.

#### Max Tree Output

An Upper bound on the absolute value of a single tree output. Used to minimize overfitting. If using FastRank for regression, increase this bound.

*On the Command Line:*

Add mo= followed by the value to the learner settings string.

#### Early Stopping Rule

This feature can decide the number of iterations based on the result of validation tests so that the training doesn't fall into overfitting. For the detail of these rules, please see this paper: *Lodwich, Aleksander, Yves Rangoni, and Thomas Breuel, "*[*Evaluation of robustness and performance of Early Stopping Rules with Multi Layer Perceptrons*](http://iupr1.cs.uni-kl.de/~shared/publications/2009-lodwich-ijcnn-robust-performance-stop-rule.pdf)*," Neural Networks, 2009.*

To enable early stopping, select one of the early stopping rules from the drop-down in the GUI and add the [validation dataset](#learning_modes_train_mode_htm_va_3838).  Possible parameter values for the rules are threshold (**TH**) and/or window size (**W**).

|  |  |  |  |
| --- | --- | --- | --- |
| Early stopping rule | Short Name | Params | When it stops |
| Tolerant | **TR** | **TH** | Stops when "validationScore" < "bestScore" - **TH** |
| Loss of Generality | **GL** | **TH** | Stops when "validationScore" < "bestScore" \* (1 - **TH**) |
| Low Progress | **LP** | **TH, W** | Stops when "recentAvg" >= (1 -**TH**) \* "recentBest" |
| Generality to Progress rate | **PQ** | **TH, W** | Stops when "validationScore" \* "recentBest" <= (1 - **TH**) \* "bestScore" \* "recentAvg" |
| Consecutive Loss in Generality | **UP** | **W** | Stops when the score degraded **W** times in a row. |

Where:

* validationScore: the score of the latest validation test.
* bestScore: the best ever validation score.
* recentBest: the best training score in the past **W** steps.
* recentAvg: the average training score in the past **W** steps.

*On the Command Line:*

* Add esr= followed by the early stopping rule and paramaters to the learner settings string
* Add esmt= followed by the number that indicates the early stopping metric to the learner settings string.

You can choose the early stopping metric:

* For FastTree classification, error rate (0) is always used.
* For FastTree regression,  L1 (1) is the default or select L2 (2)
* For FastTree ranking, NDCG@1 (1) is the default or select NDCG@3 (3)
* For Neural Nets mean error of NN outputs (0) is always used

#### Enable Pruning, Use Tolerant Pruning, Pruning Threshold, Pruning Window Size

Closely related to early stopping is pruning. It too evaluates performance per iteration on a held out validation set. The difference is, instead of stopping learning early, the learner will train all iterations, but after the fact, prune to the tree it seems best. (So, if you train 200 iterations, and iteration 190 was best, the last ten trees are discarded.) Note that pruning will use the same metric as defined by early stopping, and it has the same interpretation, so use the Early Stopping Metric to set the metric you wish to use for pruning.

If the flag Use Tolerant Pruning is disabled, the iteration that scored best on the validation set the point at which pruning happens. If it is enabled, the algorithm chooses the first tree which is within a tolerance fraction indicated by Pruning Tolerance Fraction from the best validation set score. The score is computed as a moving average with window determined by Pruning Window Size.

* Add pruning=+ to turn on pruning.
* Add prtol=+ to the learner settings string to indicate tolerant pruning.
* Add prws= followed by the window size to the learner settings string
* Add prtol= followed by the tolerance fraction to the learner settings string

#### Label Gains (only for ranking)

Indicate the [gains associated with each label](#introduction_ranking_htm_measuri_2044). To change the gains used, enter them, starting with the gain associated with label 0, comma-separated, and ending with a comma.

*On the Command Line:*

Add gains= followed by the gains string to the learner settings string.

#### Train DCG (only for ranking)

A flag indicating training using [DCG](#introduction_ranking_measuring_t_1354) instead of NDCG.

*On the Command Line:*

Add dcg=+ to the learner settings string to indicate the use of DCG.

### Weighting examples

FastRank supports weighting examples. To enable this option, weights should be specified in the input data file, and passed to the parser in the [Weight Column Idx](#input_data_format_text_input_for_5430) text box. Weights are taken into account in gradient computation and in finding best tree splits. They are not taken into account for the purpose of calculating the [number of instances in each leaf](#tlc_learners_fasttree_htm_min_do_2896). Using weighted examples implies also that metrics are computed both weighted and unweighted. The latter being often useful in better weight impact understanding.

## Parallel Ensemble

### Overview

An Ensemble is a set of models, each trained on a sample of the training set. In some cases, training an ensemble instead of a single model is used to boost the accuracy of a given algorithm. Parallel Ensemble is an umbrella word to denote all types of ensembles where the models are independent and can run in parallel. Here are some examples:

* Bagging (**b**ootstrap **agg**regat**ing**). With this method, image002.png new training sets of smaller size are generated by sampling from the training set with replacement and the outputs of the models are combined by voting. For more information, see the [Wikipedia entry for Bootstrap aggregating](http://en.wikipedia.org/wiki/Bootstrap_aggregating), or the paper [Bagging Predictors](http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.32.9399) by L. Breiman.
* Stacking. With this method, the models are trained independently and their outputs are combined by training a new model having the outputs of the Ensemble models as input.
* Random Forest. With this method, independent random trees are created by sampling instances and features.

The quality of an Ensemble depends on two factors; Accuracy and Diversity. Ensemble can be analogous to Teamwork. If every team member is diverse and competent, then the team can perform very well. Here a team member is a Base learner and the team is the Ensemble.

The accuracy depends on the selection of the base learners. In order for the Ensemble to be able to boost the accuracy of the base learners, the prediction of each base learner should be better than a random prediction.

There are various methods to improve the diversity of the Ensemble:

* Use heterogeneous base learners.
* Use homogeneous base learners: The algorithm used is the same, but the parameter settings are different (using parameter sweeps).
* Use unstable base learners (such as trees).
* Use different types of [sampling mechanisms](#tlc_learners_parallel_ensemble_h_5441) to induce base learners. Three instance and two Feature sampling mechanisms are available. Together, there are six combinations available.
* Use feature sampling. If the base learner is unstable, training is highly dependent upon the features.
* Another way to create even more sampling mechanisms is by creating an Ensemble of Ensembles.
* Use [Best Diverse Selector](#tlc_learners_parallel_ensemble_h_5780) as a pruning algorithm to remove similar base learners after training.

TLC also implements a multi-class version of Parallel Ensemble, which has the same settings as the binary classification version.

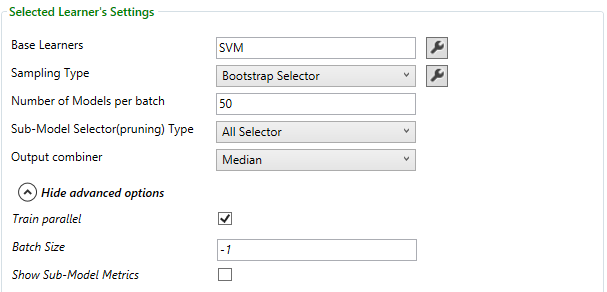
### Settings

*On the Command Line:*

The command-line argument for Parallel Ensemble is /cl WeightedEnsemble.

The command-line argument for the multi-class version is /cl WeightedEnsembleMulticlass.

After selecting Ensemble from the learners drop list, the algorithm’s settings are displayed in the right pane of the TLC GUI.



#### Base Learners

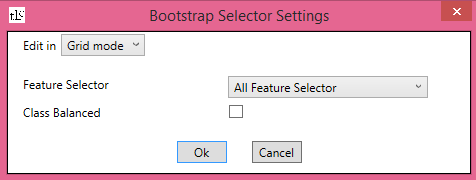
The algorithms used to train each model. Clicking the image004.png button opens a new window for selecting the algorithms. Either choose one algorithm to train all the models, use the same algorithm with different parameter settings using parameter sweeps (this is known as a Homogeneous Ensemble), or choose a combination of different algorithms and parameter settings (this is known as a Heterogeneous Ensemble).The default base algorithm is SVM (Pegasos-Linear).

*On the Command Line:*

* Add bp= followed by the binary classification algorithm name to the learner settings string. All the binary classification algorithms available in TLC can be used with Parallel Ensemble
* The algorithm name is followed by the settings string inside a pair of curly brackets
* To use more than one algorithm, enter each algorithm name and its settings preceded by bp=

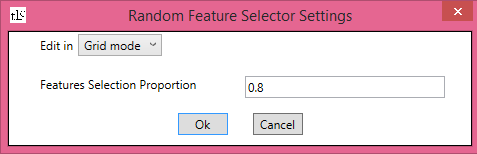
#### Sampling Type

This drop list contains the available Subset Selectors that control how the training samples are created. There are three available methods. The default method is Bootstrap Selector. The second method is Random Partition Selector. In this method, the training set is randomly partitioned into subsets. The last method is All Selector, where every model is trained using the whole training set. Each one of these Subset Selectors has two options for selecting features: either all the features are used (this is the default method), or a random subset of the features is selected for each model. Choose the Feature Selector (and other parameters that can be set for specific Subset Selectors) by clicking the image004.png button next to the Subset Selector drop list. Clicking this button opens a dialog box to specify the relevant parameter values:



Alternatively, the parameter values can be passed using a *subset selector settings string*, by choosing “Text mode” from the drop list on the top of the dialog box  and writing the subset selector settings string in the format parameter1=value1 parameter2=value2 ….

If the Random Feature Selector is chosen, click the image004.png button next to it to choose the fraction of features to be selected. A dialog box to specify the Feature Selection Proportion is displayed:



The default value of this fraction is 0.8. Alternatively, this parameter can be passed using a *feature selector settings string*, by choosing “Text mode” from the drop list on the top of the dialog box and writing the feature selector settings string in the format parameter1=value1 parameter2=value2 ….

*On the Command Line:*

* Add st=RandomPartitionInstancesSelector, st=AllDatasetSelector or st=BootstrapSelector to the learner settings string
* Pass the subset selector settings string to the program in a pair of curly brackets
* To choose a random subset of features, add fs=RandomFeatureSelector { fp=value } to the subset selector settings string, where value is the fraction of features to be selected

##### Random Partition Selector

In this method the training set is randomly partitioned into parts, where the number of parts is as indicated in the [Num Models per batch](#tlc_learners_parallel_ensemble_h_7993) text box.

##### Bootstrap Selector

This is the default sampling method. In this method, the sub-training sets are created by sampling uniformly with replacement from the training set. In addition to choosing the feature selection method, the settings of this selector include a check box to indicate that minority classes should be over sampled in order to create balanced subsets.

*On the Command Line:*

To indicate over sampling minority classes, add cb=+ to the sampling selector settings string.

##### All Instance Selector

Use the whole training set for each model.

#### Number of Models per batch

Indicates the number models to train, i.e., the number of subsets of the training set to sample. The default value is 50. If [batches](#tlc_learners_parallel_ensemble_h_7993) are used then this is the number of models per batch.

*On the Command Line:*

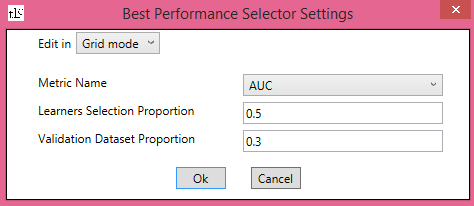
Add nm= followed by the number of models to the learner settings string.

#### Sub-Model Selector (pruning) Type

Determines the efficient set of models the Output Combiner uses, and removes the least significant models. This is used to improve the accuracy and reduce the model size. This is also called pruning. There are two major types of pruning algorithms, the first using the performance of each model and the other using the diversity among models. There is also an option to do no model pruning.

Each Sub-Model Selector has its own user defined parameters that can be specified by clicking the image004.png button on the right of the Selector’s name.

Clicking this button opens a dialog box to specify the relevant parameter values:



Alternatively, the parameter values can be passed using a *submodel selector settings string*, by choosing “Text mode” from the drop list on the top of the dialog box  and writing the submodel selector settings string in the format parameter1=value1 parameter2=value2 ….

*On the Command Line:*

* Add pt=AllSelector, pt=BestPerformanceSelector or pt=BestDiverseSelector to the learner settings string
* Pass the submodel selector settings string to the program after the Sub-Model Selector name, in a pair of curly brackets

##### All Selector

Combine all the models to create the output. This is the default submodel selector.

##### Best Performance Selector

Combine only the models with the best performance. The settings for this Sub-Model Selector include the metric by which to calculate a model’s performance (the default is AUC), the Learners Selection Proportion (a number between 0 and 1 that determines the fraction of learners used by the output combiner, the default is 0.5) and the Validation Dataset Proportion (a number between 0 and 1 that determines what fraction of the training set is used to compute the chosen metric for each model. The rest of the training set is used for training. The default value is 0.3. If it is changed to 0, the whole training set is used both for training and for this computation).

*On the Command Line:*

* To choose the metric, add mn= followed by the metric name to the submodel selector settings string. The options are:
  + For binary classification: Accuracy, PosPrecision, PosRecall, NegPrecision, NegRecall, LogLoss and LogLossReduction and AUC. The default is AUC.
  + For multi-class classification: AccuracyMicroAvg,AccuracyMacroAvg, LogLoss and LogLossReduction. The default is AccuracyMicroAvg.
  + For Regression: L1, L2 and RMS.  The default is L1.
* To choose the learners selection proportion, add lp= followed by its value to the submodel selector settings string (the default is 0.5)
* To choose the validation dataset proportion, add vp= followed by its value to the submodel selector settings string (the default is 0.3)

##### Best Diverse Selector

Combine models whose predictions that are diverse as possible. The settings for this Sub-Model Selector include the diversity measure type (currently there is only one option, Disagreement Diversity Measure), the Learners Selection Proportion (a number between 0 and 1 that determines the fraction of learners used by the output combiner, the default is 0.5) and the Validation Dataset Proportion (a number between 0 and 1 that determines what fraction of the training set is used to compute the chosen metric for each model. The rest of the training set is used for training. The default value is 0.3. If it is changed to 0, the whole training set is used both for training and for this computation).

*On the Command Line:*

* To choose the learners selection proportion, add lp= followed by its value to the submodel selector settings string (the default is 0.5)
* To choose the validation dataset proportion, add vp= followed by its value to the submodel selector setting string( the default is 0.3)

#### Output Combiner

Indicates how to combine the predictions of the different models into a single prediction.

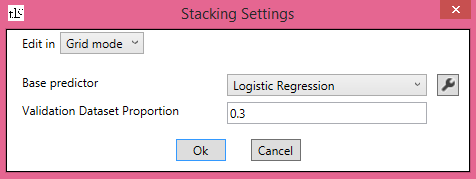
*On the Command Line:*

Add oc=Stacking or oc=Voting to the learner settings string.

There are five available Output Combiners:

##### Stacking Combiner

This Output Combiner computes the output by training a model on a training set where each instance is a vector containing the outputs of the different models on a training instance, and the instance’s label. The settings of the Output Combiner can be changed by the user by clicking the image004.png button. Clicking this button opens a dialog box to specify the relevant parameter values:



Alternatively, the parameter values can be passed using a *combiner settings string*, by choosing “Text mode” from the drop list on the top of the dialog box  and writing the combiner settings string in the format parameter1=value1 parameter2=value2 ….

The parameters that can be defined by the user are the algorithm that trains this model (the default algorithm is Logistic Regression) and the fraction of the training set used to train this model (The rest of the training set is used for training the ensemble. The default value is 0.3. If it is changed to 0, the whole training set is used both for training and for this computation).

*On the Command Line:*

* To choose a learning algorithm other than Logistic Regression, add bp= followed by the algorithm name to the combiner settings string
* To change the validation dataset proportion, add vp= followed by its value to the combiner settings string
* The combiner settings string is passed to the program after the Output Combiner name (Stacking) in a pair of curly brackets

##### Average

This output combiner computes the average of the outputs of the trained models.

##### Voting

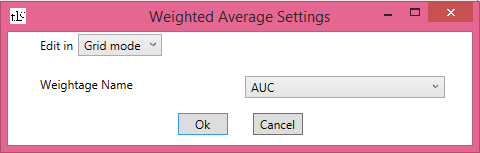
This output combiner computes the fraction of positive predictions and the fraction of negative predictions among the different models, and outputs the larger number, with the corresponding sign.

##### Median

This output combiner computes the median of the outputs of the trained models.

##### Weighted Average

This output combiner computes the weighted average of the outputs of the trained models. The settings of the Output Combiner can be changed by the user by clicking the image004.png button. Clicking this button opens a dialog box to specify the relevant parameter values:



Alternatively, the parameter values can be passed using a *combiner settings string*, by choosing “Text mode” from the drop list on the top of the dialog box  and writing the combiner settings string in the format parameter1=value1 parameter2=value2 ….

The parameter of the Weighted Average output combiner is Weightage Name, which indicates the weight to be used for each model. The options are the different metrics available.

#### Train Parallel

This check box is checked by default, meaning that all the base learners run asynchronously.

**Tip:** If the base learner is Parallel Ensemble, it is recommended to uncheck this check box to control the excessive number of threads.

*On the Command Line:*

To disable asynchronous learning, add tp=- to the learner settings string.

#### Batch Size

Train the models iteratively on subsets of the training set of this size. When using this option, it is assumed that the training set is randomized enough so that every batch is a random sample of instances. The default value is -1, indicating using the whole training set. If the value is changed to an integer greater than 0, the number of trained models is the number of batches (the size of the training set divided by the batch size), times the [number of models per batch](#tlc_learners_parallel_ensemble_h_7993). This option is used to train extremely large datasets, or when the RAM is small.

*On the Command Line:*

Add bs= followed by the number of models to the learner settings string.

#### Show Sub-Model Metrics

By default this check box is unchecked. If it is checked, the metrics of every model in the Ensemble are printed to the Standard Output. The metrics are calculated on a validation set specified by the [Sub-Model Selector type](#tlc_learners_parallel_ensemble_h_8474) or the [Stacking combiner](#tlc_learners_parallel_ensemble_h_4432), or the training instances if a validation set is not available.

*On the Command Line:*

Add sm=+ to the learner settings string to indicate showing the sub-model metrics.

## Local Deep SVM

### Overview

LD-SVM learns a binary, non-linear SVM classifier with a kernel that is specifically designed to reduce prediction time. LD-SVM learns decision boundaries that are locally linear. This implies that a test point can be efficiently classified by testing it against its local decision boundary rather than the entire set of decision boundaries all over feature space. Furthermore, LD-SVM brings down the cost of testing a point against its local decision boundary to be logarithmic in the number of training points. This implies that LD-SVM is exponentially faster at prediction than traditional SVMs, such as the RBF-SVM, which need to compute the kernel over all support vectors in order to make a prediction.

LD-SVM is most useful when all the following conditions are met:

* You have a binary classification problem or you can reduce your problem to a set of binary classification problems.
* Linear classifiers are not performing well
* Non-linear SVMs or other classifiers yield high classification accuracies but are taking too much time for making predictions.
* You can afford to sacrifice prediction accuracy in order to bring down prediction time.

LD-SVM should not be used in situations where linear classifiers are already giving good results or when high classification accuracies can be achieved by adding small amounts of non-linearity (such as with a neural network with a small number of hidden nodes).

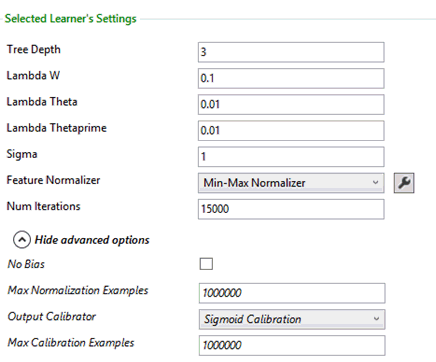
More details about LD-SVM can be found in this [paper](http://research.microsoft.com/en-us/um/people/manik/pubs/Jose13.pdf), [slide deck](http://research.microsoft.com/~manik/talks/ICML13.pdf) and [talk](http://resnet/fullvideo.aspx?id=31297).

### Settings

*On the Command Line:*

The command-line argument for Local Deep SVM is /cl LDSVM.

After selecting Local Deep SVM from the learners drop list, the algorithm’s settings are displayed in the right pane of the TLC GUI.



#### Tree Depth

Depth of the LDSVM tree. The default value is 3. The cost of prediction increases linearly with tree depth and therefore the depth should be chosen based on the prediction budget. Training time should roughly double as the depth is increased by one. Prediction accuracy should increase, reach a peak and then decrease with increasing depth.

*On the Command Line:*

Add depth= followed by the value to the learner settings string.

#### Lambda W

Regularizer for classifier parameter W (see equation 11 in [paper](http://research.microsoft.com/en-us/um/people/manik/pubs/Jose13.pdf)). The default value is 0.1. If 0.1 doesn’t work well you should also try {0.0001, 0.001, 0.01,}. Larger values of this parameter imply that more emphasis should be placed on maximizing the margin and less emphasis should be placed on the classification error. In order to reduce the number of parameters that should be swept over, a good rule of thumb is to set lambda W to be ten times lambda Theta.

*On the Command Line:*

Add lw= followed by the value to the learner settings string.

#### Lambda Theta

Regularizer for classifier parameter **θ** (see equation 11 in [paper](http://research.microsoft.com/en-us/um/people/manik/pubs/Jose13.pdf)). The default value is 0.01. If 0.01 doesn’t work well you should also try {0.0001, 0.001, 0.1,}. Larger values of this parameter imply that more emphasis should be placed on maximizing the margin and less emphasis should be placed on the classification error. In order to reduce the number of parameters that should be swept over, a good rule of thumb is to set lambda Theta to be one tenth of lambda W.

*On the Command Line:*

Add lt= followed by the value to the learner settings string.

#### Lambda Thetaprime

Regularizer for classifier parameter **θ’** (see equation 11 in [paper](http://research.microsoft.com/en-us/um/people/manik/pubs/Jose13.pdf)). The default value is 0.01. If 0.01 doesn’t work well you should also try {0.0001, 0.001, 0.1,}. Larger values of this parameter imply that more emphasis should be placed on maximizing the margin and less emphasis should be placed on the classification error. In order to reduce the number of parameters that should be swept over, a good rule of thumb is to set lambda Thetaprime to be one tenth of lambda W.

*On the Command Line:*

Add lp= followed by the value to the learner settings string.

#### Sigma

Parameter for sigmoid sharpness **σ** (see equation 10 in [paper](http://research.microsoft.com/en-us/um/people/manik/pubs/Jose13.pdf)). The default value is 1. If 1 doesn’t work well you should also try {0.1, 0.01, 0.001}. Larger values of this parameter imply that the tanh in local kernel image008.png is saturated whereas a smaller value imply linear operating range of  image008.png.

*On the Command Line:*

Add s= followed by the value to the learner settings string.

#### Feature Normalizer

By default, the features are normalized using the min-max normalizer. To specify no normalization, choose <None> from the Normalizer drop list. A different normalizer can also be chosen using this drop list.

*On the Command Line:*

* Add norm={} to the learner settings string to indicate no normalization
* Add norm=GaussianNormalizer or norm=BinNormalizer to the learner settings string to indicate normalization using a different normalizer. If a normalizer settings string is needed it follows the normalizer name, in a pair of curly brackets

#### Num Iterations

Indicates the number of times the algorithm updates the classifier parameters, using a random subset of examples (currently sqrtN.png, where N.png is the total number of examples) from the training set. The default value is 15,000.

*On the Command Line:*

Add iter= followed by the value to the learner settings string.

#### No Bias

By default, the TLC implementation of LD-SVM includes a bias term which is equivalent to appending a constant term to the feature vector. In some cases, the bias term may increase the accuracy of the classifier. Check the No Bias check box if you don’t want to use a bias term.

*On the Command Line:*

Add nobias=+ to the learner settings string to indicate b=0.

#### Max Normalization Examples

The maximal number of instances that the normalizer processes before computing the normalization function. The default value is 1 million. This option is used to speed up the learning in cases where there is a large number of examples.

*On the Command Line:*

Add numNorm= followed by the value to the learner settings string.

#### Output Calibrator

Some classification algorithms produce raw scores that are unbounded. In order to convert these outputs into a binary prediction (or the predicted probability of belonging to one of the classes) the output must be calibrated. By default [sigmoid calibration](#introduction_classification_outp_9719). The other two options in the Calibrator drop list are [PAV calibration](#introduction_classification_outp_1316) and [Naïve calibrator](#introduction_classification_outp_6402).

*On the Command Line:*

* Add cali=PAVCalibration or cali=NaiveCalibration to the learner settings string to indicate calibration using a different calibrator
* Add cali={}to indicate no calibration

#### Max Calibration Examples

The maximal number of instances that are processed before computing the calibration function. The default value is 5 million. This option is used to speed up the calibration process in cases where there is a large number of examples.

*On the Command Line:*

Add numCali= followed by the value to the learner settings string.

### Sample Results

LD-SVM aims to reduce prediction time. The following 3 tables demonstrate how LD-SVM’s prediction time and accuracy lie between that of TLC’s linear and non-linear classifiers. The height of the LD-SVM tree can be adjusted to get varying levels of prediction accuracy and time. The hyper-parameter settings for the different learners are summarized here:

Table 1: Results on CIFAR10Binary

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **LR** | **Pegasos** | **FastRank** | **Bramble** | **NeuralNet** | **LD-SVM** |
| **Prediction Accuracy (%)** | 69.14 | 67.15 | 78.35 | 72.87 | 69.84 | 76.54 |
| **Prediction Time (ms)** | 494 | 391 | 2260 | 2270 | 520 | 419 |
| **Training Time (s)** | 36.14 | 26.54 | 331 | 662 | 139 | 19.28 |

               Table 2: Results on LetterBinary

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **LR** | **Pegasos** | **FastRank** | **Bramble** | **NeuralNet** | **LD-SVM** |
| **Prediction Accuracy (%)** | 72.38 | 72.73 | 98.3 | 97.03 | 93.62 | 96.43 |
| **Prediction Time (ms)** | 42.78 | 24.96 | 423 | 1446 | 34.8 | 32.76 |
| **Training Time (s)** | 0.92 | 1.35 | 6.31 | 81 | 4 | 239 |

Table 3: Results on MNIST8MBinary

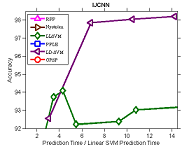
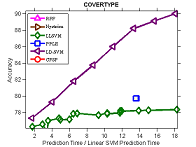
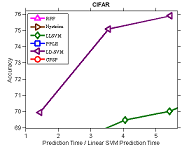
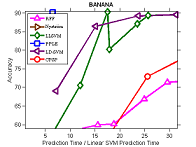
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **LR** | **Pegasos** | **FastRank** | **Bramble** | **NeuralNet** | **LD-SVM** |
| **Prediction Accuracy (%)** | 79.25 | 79.57 | 97.72 | 97.55 | 99.82 | 98.01 |
| **Prediction Time (s)** | 13.2 | 26.84 | 982.3 | 453.2 | 176 | 15.4 |
| **Training Time (s)** | 749 (24T) | 469 | 4517 (20T) | 13498 (20T) | 8442 | 3913 |

Note: T stands for number of threads

The following table and figures compare the performance of LD-SVM to an RBF-SVM as well as state-of-the-art approaches for speeding up non-linear SVM prediction. Note that all data sets represent binary classification problems and have been pre-processed to have zero mean and unit variance. We have turned off SSE instructions in LD-SVM for these experiments in order for a fair comparison with other non-SSE based techniques.

Table 4: LD-SVM can signiﬁcantly speed up prediction time over an RBF-SVM (normalized time = prediction time / linear SVM prediction time).

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Data Set** | **Classification Accuracy (%)** | | | **Prediction Time (Normalized)** | | **Prediction Speedup over**  **RBF-SVM** | **Training Time (in minutes)** | | |
| **Linear SVM** | **RBF-SVM** | **LD-**  **SVM** | **RBF-**  **SVM** | **LD-**  **SVM** | **Linear**  **SVM** | **RBF-**  **SVM** | **LD-**  **SVM** |
| BANANA | 59.18 | 90.3 | 90.02 | 1481.8 | 32.12 | 46.13 | 1.48E-05 | 0.006 | 0.01 |
| CIFAR | 69.16 | 81.62 | 76.02 | 38094.48 | 5.37 | 7081.6 | 6.76E-02 | 1283.68 | 0.278 |
| CoverType | 76.27 | 91.87 | 90.03 | 229418 | 16.67 | 13757.95 | 2.35 | 1369.96 | 7.990 |
| IJCNN | 92.2 | 98.68 | 98.31 | 6684.53 | 12.24 | 546 | 3.93E-02 | 0.45 | 0.090 |
| Letter | 73.08 | 98.1 | 96.02 | 11020 | 33.29 | 330.98 | 5.75E-04 | 0.43 | 2.200 |
| Magic04 | 79.08 | 86.65 | 86.19 | 9489.79 | 13.26 | 709.92 | 1.74E-03 | 0.17 | 0.047 |
| MNIST | 87.9 | 97.45 | 97.28 | 12130.98 | 9.37 | 1294.42 | 2.86E-01 | 39.12 | 1.376 |
| USPS | 83.65 | 96.96 | 95.63 | 3375 | 8.15 | 413.79 | 5.97E-03 | 0.75 | 0.096 |
| RCV1 | 93.18 | - | 95.27 | - | 1.18 | - | 0.13 | - | 0.5 |
| MNIST8M | 79.24 | - | 98.01 | - | 1.16 | - | 0.70 | - | 65.21 |



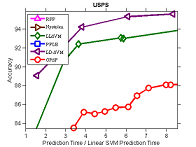
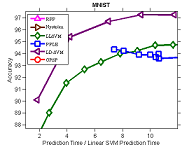
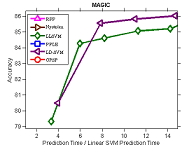
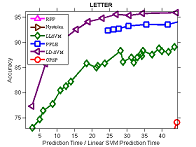
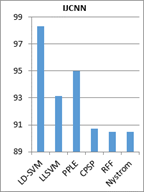
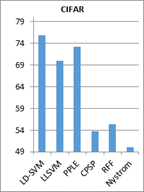
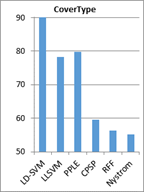
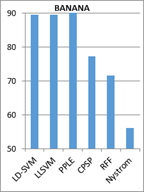


Figure 1: Plots of prediction time versus prediction accuracy for state-of-the-art methods for speeding up non-linear SVM prediction. LD-SVM can have significantly higher classification accuracies as compared to the other methods for a given prediction cost. A method's curve not appearing in the plot for a given data set indicates that, for the given prediction cost range, the methods performance was worse than that of a linear SVM. Figure best viewed magnified.



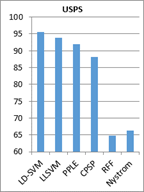
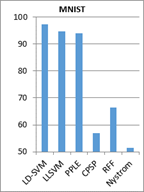
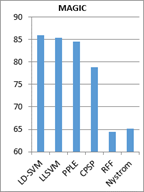
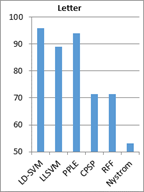


Figure 2: Bar charts of prediction accuracy for a fixed prediction cost.

The hyper-parameter settings used for the different learners are summarized here:

CIFAR10Binary

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **LR** | **Pegasos** | **FastRank** | **Bramble** | **NeuralNet** | **LD-SVM** |
| l2=0.1,  m=50 | iter=100,  initwts=1 | nl=128, mil=50,  iter=500,  lr=0.158740 | numTrees=60,  maxDepth=20,  numSamples=20 | iter=160,  lr=0.02 | depth=2, lw=0.01  lt=0.1, lp=0.1  s=0.001, norm={}  cali={} |

LetterBinary

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **LR** | **Pegasos** | **FastRank** | **Bramble** | **NeuralNet** | **LD-SVM** |
| l1=0,  l2=0,  ot=0.0001 | lambda=1e-5  iter=100,  initwts=0.1 | nl=128,  iter=500,  lr=0.1587401 | numTrees=100,  maxDepth=20,  numSamples=4 | iter=160, lr=0.1,  initwts=1 | depth=9, lw=0.1,  lt=0.01, lp=0.001,  s=1, norm={},  cali={} |

MNIST8MBinary

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **LR** | **Pegasos** | **FastRank** | **Bramble** | **NeuralNet** | **LD-SVM** |
| l1=0,  l2=0,  m=50 | lambda=1e-5  iter=100,  initwts=1 | nt=20,  iter=500,  lr=0.1 | numTrees=40,  numThreads=20,  maxMemGigs=50 | iter=30 | depth=5, lw=0.01,  lt=0.001,s=0.01,  iter=30000, norm={},cali={} |

## StratoLearner

**Note:** StratoLearner is not available by default in the learners drop list of the TLC GUI. See [below](#tlc_learners_stratolearner_htm_s_4177) for details on how to add it.

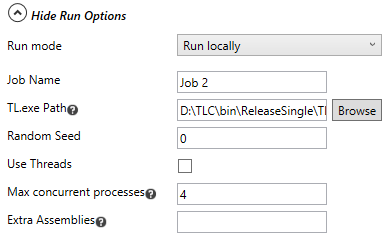
### Overview

StratoLearner is an algorithm that partitions the training dataset into separate parts, and trains one model for each part. The parts are determined by using one or more *key* name/attribute features, where in every part, all instances have the same unique value for these features. A model is trained only on parts that contain more instances than some user defined threshold. An additional default model is trained using the whole dataset. The prediction is made by examining a new instance’s value of the key features, and predicting using the model trained for this value if it exists, or the default model otherwise.

TLC also implements a regression version and a ranking version of Strato Learner, which have the same settings as the binary classification version.

### Settings

Strato Learner is not available by default in the learners drop list. To add it, StratoLearner.dll must be referenced in the “Extra Assemblies” text box in the bottom of the Advanced Options box in the left pane of the TLC GUI:



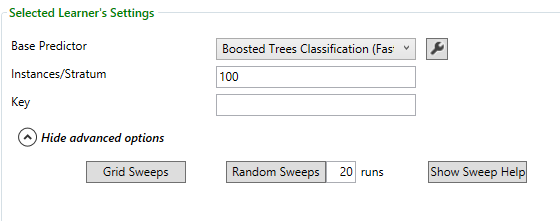
The assembly is located in the SupplementalLearners folder that is downloaded with the TLC package. The complete path of the assembly must be specified. After specifying the assembly’s path, change the task in the “Task” drop list then switch back to “Binary Classification”. StratoLearner should now be available on the learners drop list.

*On the Command Line:*

The command-line argument for StratoLearner is /cl StratoLearner.

To use Strato Learner, use the command-line argument /dll followed by the full path of StratoLearner.dll.

After selecting Strato Learner from the learners drop list, the algorithm’s settings are displayed in the right pane of the TLC GUI.



#### Base Predictor

The algorithm used to train each model. The default algorithm is FastTree. Clicking the image004.png button on the right of the algorithm name displays a dialog box where the parameters for the specific algorithm can be specified. Alternatively, the parameter values can be passed using the learner settings string by choosing “Text mode” from the drop list on the top of the dialog box  and writing the learner settings string in the format parameter1=value1 parameter2=value2 ….

*On the Command Line:*

* Add bp= followed by the binary classification algorithm name to the learner settings string. All the binary classification algorithms available in TLC can be used with Strato Learner
* The algorithm name is followed by the settings string inside a pair of curly brackets

#### Instances/Stratum

The minimum number of instances with a specific value of the key features that are needed to train a model for this value.

*On the Command Line:*

Add min= followed by the number of instances the learner settings string.

#### Key

The features that should be used as keys for partitioning the dataset. This parameter does not have a default value and must be defined by the user. Features that can be used as keys are name features, or attribute features. They are specified by n (for name features) or a (for attribute features), followed by an integer, which is the 0-based index of the name (or attribute) features, as indicated in the [Name Columns (or Attribute Columns)](#input_data_format_text_input_for_1375) text box. If there is more than one key feature, list them separated by commas. For example, if the name features are in columns 3, 5, 9 and 15 and the attribute features are in columns 17 and 23, then if n3,a1 is specified as the key, the instances are partitioned by the values in columns 15 and 23 (the 4th name feature column and the 2nd attribute feature column).

*On the Command Line:*

Add strat= followed by the list of keys to the learner settings string to pass it to the program.

## One-vs-all / pairwise coupling

### Overview

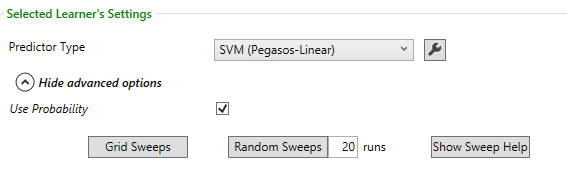
For an overview of these techniques, [go here](#introduction_classification_mult_4439).

### Settings

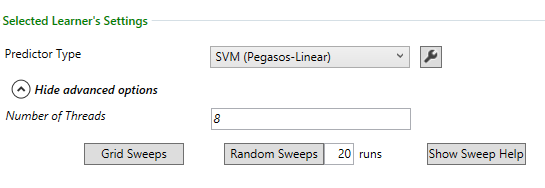
*On the Command Line:*

* The command-line argument for OVA is /cl OVA.
* The command-line argument for PKPD is /cl PKPD.

After selecting [One-vs-All](#introduction_classification_mult_4975) or [Pairwise Coupling (PKPD)](#introduction_classification_mult_7939) from the learners drop list, the algorithm’s settings are displayed in the right pane of the TLC GUI.



One-Vs-All



Pair-Wise-Coupling

#### Predictor Type

The “Predictor Type” parameter indicates the base algorithm that is used to train the binary classifiers. The default algorithm is SVM (Pegasos-Linear). Clicking the image003.png button on the right of the binary classification algorithm name displays a dialog box where the parameters for the specific algorithm can be specified. Alternatively, the parameter values can be passed using the learner settings string by choosing “Text mode” from the drop list on the top of the dialog box  and writing the learner settings string in the format parameter1=value1 parameter2=value2 ….

*On the Command Line:*

* Add p= followed by the binary classification algorithm name to the learner settings string. All the binary classification algorithms available in TLC can be used with OVA or PKPD
* The algorithm name is followed by the settings string inside a pair of curly brackets

#### Use Probability (for One-Vs-All)

By default, the One-Vs-All algorithm uses the probability returned by the classifiers to identify the top-score class. If the “Use Probability” check box is unchecked, the uncalibrated results are used instead.

*On the Command Line:*

Add useprob=- to the learner settings string to indicate returning the uncalibrated results

#### Number of Threads (for PKPD)

The number of threads. Should be equal to the number of cores on the machine.

*On the Command Line:*

Add nt= followed by the number of threads to the learner settings string

## Multi-class image patching

### Overview

Patching is a regularization procedure applicable to neural nets, which has applications for training and testing.  During training, instead of a training image being passed through directly to the learner, a sub-image of the image is selected uniformly randomly among all possible sub-images of a specified dimension (no larger than the original image) and optionally flipped either horizontally or vertically. For testing, five specific sub-images are created (the four corners and the center), and the user can choose to include also the flipped images of these five sub-images. The output is the average of the outputs of the neural net (or other base learner used) on these sub-images.

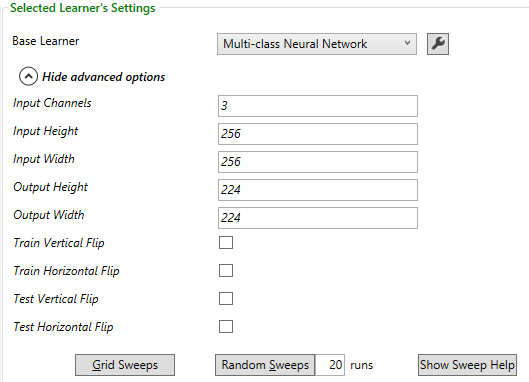
Whether flipping is appropriate or not depends heavily on the task and how dimension independent something is.  OCR like tasks would probably work better with no flipping. Typical photo classification may be aided by horizontal flipping. For tasks involving satellite images, it may be that directionality is meaningless, so perhaps both horizontal and vertical flipping is appropriate.

### Settings

*On the Command Line:*

The command-line argument for Multi-ClassImage Patching is /cl PatchMC.

After selecting Multi-Class Image Patching from the learners drop list, the algorithm’s settings are displayed in the right pane of the TLC GUI.



#### Base Learner

This parameter specifies the base learner used for training. The default algorithm is Multi-Class Neural Networks. Clicking the image002.png button on the right of the multi-class classification algorithm name displays a dialog box where the parameters for the specific algorithm can be specified. Alternatively, the parameter values can be passed using the learner settings string by choosing “Text mode” from the drop list on the top of the dialog box  and writing the learner settings string in the format parameter1=value1 parameter2=value2 ….

*On the Command Line:*

* Add bp= followed by the multi-class  classification algorithm name to the learner settings string. All the multi-class classification algorithms available in TLC can be used with Parallel Ensemble
* The algorithm name is followed by the settings string inside a pair of curly brackets

#### Input Channels, Input Height/Width

Since instances do not retain information on the geometry of the image, these parameters must be specified. The assumption is that the features are ordered first by channel, then by row and finally by column. For example, if the image is an RGB image, then the feature vector contains first the red pixels ordered by rows, then the green pixels and then the blue pixels.

The Input Channels parameter specifies the number of channels. It is typically either 3 (RGB) or 1 (greyscale/B&W).

The Input Height and Input Width parameters specify the height and width of the input images.

*On the Command Line:*

* Add inchans= followed by the number of channels to the learner settings string
* Add inheight= followed by the height of the input images to the learner settings string
* Add inwidth= followed by the width of the input images to the learner settings string

#### Output Height/Width

These parameters specify the height and width of the sampled patched output images.

*On the Command Line:*

* Add outheight= followed by the height of the sampled patched output images to the learner settings string
* Add outwidth= followed by the width of the sampled patched output images to the learner settings string

#### Train Vertical/Horizontal Flip

These parameters specify whether input images should be vertically or horizontally flipped vertically with probability ½ during training. By default these checkboxes are unchecked.

*On the Command Line:*

* Add trvflip=+ to the learner settings string to indicate vertical flipping with probability 1/2 during training
* Add trhflip=+ to the learner settings string to indicate horizontal flipping with probability 1/2 during training

#### Test Vertical/Horizontal Flip

Specifies whether the vertically or horizontally flipped image should be included in the averaging during testing. By default these checkboxes are unchecked, in which case 5 sub-images are averaged over. If one of the checkboxes is checked then 10 sub-images are averaged over, and if both are checked then 20 sub-images are averaged over.

*On the Command Line:*

* Add tevflip=+ to the learner settings string to indicate vertical flipping with probability 1/2 during testing
* Add tehflip=+ to the learner settings string to indicate horizontal flipping with probability 1/2 during testing

## Poisson regression

### Overview

Poisson Regression assumes the unknown function, denoted image002.png has a Poisson distribution, i.e., given the instance image004.png, for every image006.png, the probability that its value is image008.png is

image010.png

where image012.png.

Given the set of training examples, the algorithm tries to find the optimal values for image014.png by trying to maximize the log likelihood of the parameters given the input. The likelihood of the parameters  image014.png is the probability that the training data was sampled from a distribution with these parameters.

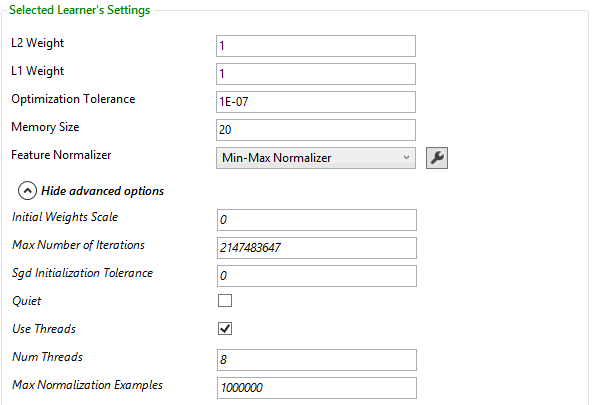
For more information, see the [Wikipedia entry for Poisson regression](http://en.wikipedia.org/wiki/Poisson_regression).

### Settings

*On the Command Line:*

The command-line argument for Poisson Regression is /cl PoissonRegression.

After selecting Poisson Regression from the learners drop list, the algorithm’s settings are displayed in the right pane of the TLC GUI.



#### L1 Weight, L2 Weight

This learner can use a linear combination of L1 and L2 regularizations. The default value for both coefficients is 1.

*On the Command Line:*

Add l1= or l2= followed by their values to the learner settings string.

#### Optimization Tolerance

This parameter sets the threshold for the optimizer convergence. In other words, if the improvement between iterations is less than the threshold, the algorithm stops and returns the current model. The default value of this parameter is 1E-07.

*On the Command Line:*

Add ot= followed by the threshold to the learner settings string.

#### Memory Size

The technique used for optimization here is L-BFGS, which uses only a limited amount of memory to compute the next step direction. This parameter indicates the number of past positions and gradients to store for the computation of the next step. The default value of this parameter is 20.

*On the Command Line:*

Add m= followed by the value to the learner settings string.

#### Feature Normalizer

By default, the features are normalized using the min-max normalizer. To specify no normalization, choose <None> from the Normalizer drop list. A different normalizer can also be chosen using this drop list.

*On the Command Line:*

* Add norm={} to the learner settings string to indicate no normalization
* Add norm=GaussianNormalizer or norm=BinNormalizer to the learner settings string to indicate normalization using a different normalizer. If a normalizer settings string is needed it follows the normalizer name, in a pair of curly brackets

#### Initial Weights Scale

The initial weights are initialized randomly. This parameter indicates the diameter of their initialization, i.e., if the diameter is specified to be d, the weights are uniformly distributed between -d/2 and d/2. The default value is 0, indicating all weights are initialized to 0.

*On the Command Line:*

* Add initwts= followed by its value to the learner settings string
* initwts=0 initializes all the weights to 0

#### Max Number of Iterations

Max number of iterations indicates the maximum number of optimization steps the algorithm should make. After this number of steps, the algorithm stops even if it hasn’t reached convergence.

*On the Command Line:*

Add maxiter= followed by the maximum number of iterations to the learner settings string.

#### Sgd Initialization Tolerance

If the value is changed to a number greater than 0, SGD is used to find the initial parameters, converging to the tolerance indicated.

*On the Command Line:*

Add sgd= followed by the value to the learner settings string.

#### Quiet

If this check box is checked, no output is produced during training.

*On the Command Line:*

Add q=+ to the learner settings string to indicate no output during training.

#### Use Threads, Num Threads

The “Use Threads” check box indicates whether or not to use threads, and Num thread is the number of threads (should be equal to the number of cores on the machine).

*On the Command Line:*

* Add t=- to the learner settings string to indicate no threads
* Add nt= followed by the number of threads to the learner settings string

#### Max Normalization Examples

The maximal number of instances that the normalizer processes before computing the normalization function. The default value is 1 million. This option is used to speed up the learning in cases where there is a large number of examples.

*On the Command Line:*

Add numNorm= followed by the value to the learner settings string.

## Stochastic gradient descent

### Overview

Stochastic gradient descent is an optimization method used to train a wide range of models in machine learning. In the TLC implementation of SGD, it is for linear regression. In other words, the algorithm finds a good approximation for an unknown function of the form image002.png, where image004.png is a weights vector and image006.png is the feature vector.

The TLC implementation of SGD is an online implementation, i.e., it processes the instances in the training set one at a time. In every step of the optimization, the algorithm examines the next example from the training set, and changes the weights in a way that the loss function on the chosen example is decreased.

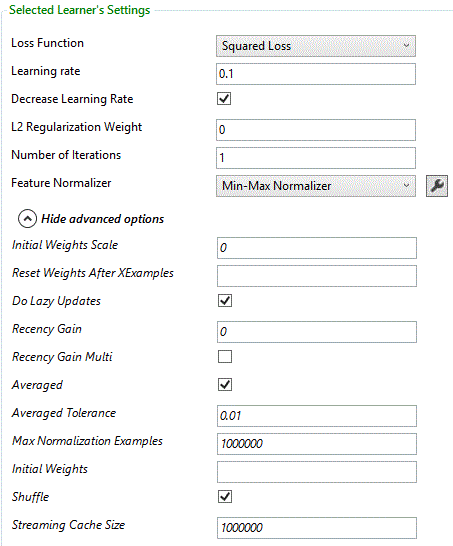
For more information, see the [Wikipedia entry for Stochastic Gradient Descent](http://en.wikipedia.org/wiki/Stochastic_gradient_descent) and the [Wikipedia entry for Linear Regression](http://en.wikipedia.org/wiki/Linear_regression).

### Settings

*On the Command Line:*

The command-line argument for Stochastic gradient descent is /cl SGD.

After selecting SGD from the learners drop list, the algorithm’s settings are displayed in the right pane of the TLC GUI.



#### Learning Rate

Determines the size of the step taken in the direction of the gradient in each step of the learning process. The default learning rate is 0.1.

*On the Command Line:*

Add lr= followed by the learning rate to the learner settings string.

#### Decrease Learning Rate

A flag indicating whether to reduce the learning rate as iterations progress. By default, the learning rate is not decreased. If the flag is turned on, then in iteration number image002.png the learning rate is reduced by a factor of image006.png. To indicate learning rate reductions check the “Decrease Learning Rate” check box.

*On the Command Line:*

Add decreaselr=+ (to indicate learning rate reductions) or decreaselr=- (to indicate no learning rate reductions) to the learner settings string.

#### Loss function

The default loss function is squared loss. The other losses available are Poisson loss, hinge loss and exponential loss.

*On the Command Line:*

Add loss= followed by the name of the loss to the learner settings string. Available losses are PoissonLoss, HingeLoss and ExpLoss.

#### L2 Regularization Weight

The weights can be L2 regularized. By default, the weight of the regularization is 0.

*On the Command Line:*

Add reg= followed by the regularization coefficient to the learner settings string.

#### Number of iterations

The number of iterations on the full training set. The default value for Neural Networks is 100 and for Averaged Perceptron and SGD it is 1.

*On the Command Line:*

Add iter= followed by the number of iterations to the learner settings string.

#### Feature Normalizer

By default, the features are normalized using the min-max normalizer. To specify no normalization, choose <None> from the Normalizer drop list. A different normalizer can also be chosen using this drop list.

*On the Command Line:*

* Add norm={} to the learner settings string to indicate no normalization
* Add norm=GaussianNormalizer or norm=BinNormalizer to the learner settings string to indicate normalization using a different normalizer. If a normalizer settings string is needed it follows the normalizer name, in a pair of curly brackets

#### Initial Weights Scale

The initial weights are initialized randomly. This parameter indicates the diameter of their initialization, i.e., if the diameter is specified to be d, the weights are uniformly distributed between -d/2 and d/2. The default value is 0, indicating all weights are initialized to 0.

*On the Command Line:*

* Add initwts= followed by its value to the learner settings string
* initwts=0 initializes all the weights to 0

#### Reset Weights After XExamples

Indicates the number of examples after which the weight vector is reset to the current weighted average.

*On the Command Line:*

Add numreset= followed by the value to the learner settings string.

#### Do Lazy Updates

Indicates that instead of updating averaged weights on every example, only update them when the loss is non-zero.

*On the Command Line:*

Add lazy=- to the learner settings string to indicate updating the weights on every example.

#### Recency Gain, Recency Gain Multi

If the value of the Recency Gain parameter is positive, it indicates that bigger weights should be given to more recent updates of the weight vector. In this case, the Recency Gain Multi check box should be checked if the recency gain should be multiplicative, and unchecked if it should be additive.

*On the Command Line:*

* Add rg= followed by the value of the recency gain to the learner settings string
* Add rgm=+ to the learner settings string to indicate multiplicative gain (default is additive)

#### Averaged

Unchecking this check box turns off averaging of the weight vectors of the different iterations.

*On the Command Line:*

Add avg=- to the learner settings string to turn off averaging.

#### Averaged Tolerance

A tolerance between 0 and 1, indicating the degree of tolerated inexactness in the averaging. Higher values will lead to faster execution of learning. This only has an effect if “averaged” is turned on.

*On the Command Line:*

Add avgtol= followed by the value to the learner settings string.

#### Initial Weights

A string containing the initial weights of the weight vector to be used by the algorithm. It is a comma-separated list of the weights.

*On the Command Line:*

Add initweights= followed by the initial weights string to the learner settings string.

**Tip:** This option can be used to output feature values. To do this, pass “0,0,0,…,1,0,…” as the initial weights string (with 1 in the location of the feature to output and 0 everywhere else), uncheck the “Calibrate Output” check box, and change the number of iterations to 0.

#### Shuffle

This check box is checked by default, and indicates that the instances are shuffled before each training iteration.

*On the Command Line:*

Add shuf=- to the learner settings string to indicate no shuffling.

#### Streaming Cache Size

This parameter indicates the size of the cache when trained in Scope. The default value is 1000000.

*On the Command Line:*

Add cache= followed by its value to the learner settings string to indicate the size of the cache when trained in Scope.

## One Class SVM

### Overview

One-class SVM is an algorithm for *anomaly detection*. Anomaly detection is the task of binary classification, when the training set contains only examples of one class. One-class SVM tries to find image002.png (image004.png is the number of training examples) that minimize the function

image006.png

subject to the conditions image008.png and image010.png. Here the image012.png’s are feature vectors, image014.png is a kernel function and image016.png is a user defined parameter.

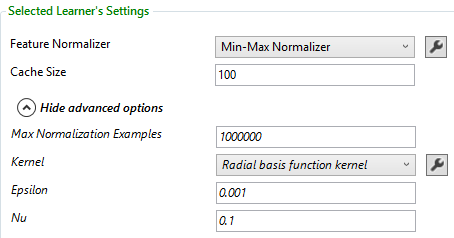
The training part is done using [libsvm](http://www.csie.ntu.edu.tw/~cjlin/libsvm/index.html). For more information, see the paper [Estimating the Support of a High-Dimensional Distribution](http://research.microsoft.com/pubs/69731/tr-99-87.pdf) by B. Schӧlkopf et al., or [New Support Vector Algorithms](http://www.stat.purdue.edu/~yuzhu/stat598m3/Papers/NewSVM.pdf) by B. Schӧlkopf et al.

### Settings

*On the Command Line:*

The command-line argument for one-class SVM is /cl OneClassSvm.

After selecting One Class SVM from the learners drop list, the algorithm’s settings are displayed in the right pane of the TLC GUI.



#### Feature Normalizer

By default, the features are normalized using the min-max normalizer. To specify no normalization, choose <None> from the Normalizer drop list. A different normalizer can also be chosen using this drop list.

*On the Command Line:*

* Add norm={} to the learner settings string to indicate no normalization
* Add norm=GaussianNormalizer or norm=BinNormalizer to the learner settings string to indicate normalization using a different normalizer. If a normalizer settings string is needed it follows the normalizer name, in a pair of curly brackets

#### Cache Size

The maximal size of the cache that stores the training data, in MB. The default value is 100. Increase this for large training sets.

*On the Command Line:*

Add cache= followed by the value to the learner settings string.

#### Max Normalization Examples

The maximal number of instances that the normalizer processes before computing the normalization function. The default value is 1 million. This option is used to speed up the learning in cases where there is a large number of examples.

*On the Command Line:*

Add numNorm= followed by the value to the learner settings string.

#### Kernel

This parameter determines the kernel that is used for training. There are currently four different kernels that can be used:

         Linear kernel: image019.png

         Polynomial kernel: image021.png

         RBF kernel: image023.png

         Sigmoid kernel: image025.png

For kernels that have user defined parameters (all except for the linear kernel), click the http://tlc/doc/images/wrench.pngbutton next to the kernel name to specify the parameter values. If the parameter g for RBF and Sigmoid kernels or a for polynomial kernel is not specified, its default value is determined as 1 / the number of features.

*On the Command Line:*

* Add ker= followed by the kernel name to the learner settings string
* The kernel settings are passed in the kernel settings string after the kernel name, in the format { parameter1=value1 parameter2=value2 … }
* For linear kernel, write ker=LinearKernel
* For polynomial kernel, write ker=PolynomialKernel. Its parameter names are a, b and d
* For RBF kernel, write ker=RbfKernel. Its parameter name is g
* For Sigmoid kernel, write ker=SigmoidKernel. Its parameter names are g and c.

#### Epsilon

This parameter sets the threshold for the optimizer convergence. In other words, if the improvement between iterations is less than the threshold, the algorithm stops and returns the current model. The default value of this parameter is 0.001.

*On the Command Line:*

Add eps= followed by the threshold to the learner settings string.

#### Nu

This parameter is described in the definition of the optimization problem in the overview of this algorithm. It determines the trade-off between the fraction of outliers and the number of support vectors The default value is 0.1.

*On the Command Line:*

Add nu= followed by the value to the learner settings string.

## Fast Forest

### Overview

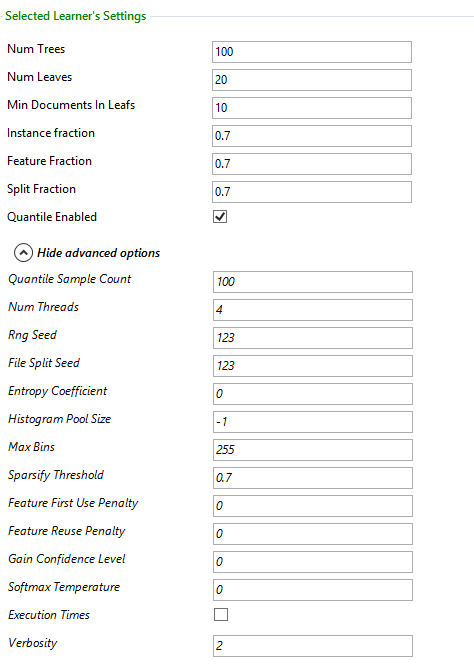
Fast forest regression is a random forest and Quantile regression forest implementation using regression Tree learner created for Fast Tree. Please refer [Random Forest](http://en.wikipedia.org/wiki/Random_forest) in Wikipedia and [Quantile regression forest](http://jmlr.org/papers/volume7/meinshausen06a/meinshausen06a.pdf) for more details. Unlike random forest which averages out the output, Quantile regression forest keeps all the labels in trees specified by “quantile sample count” and outputs the distribution so that the user can have quantile values for the given instance. Weights on instances is supported starting in TLC version 2.7.

### Settings

*On the Command Line:*

* The command-line argument for FastForest regression is /cl FastForestRegression
* The command-line argument for FastForest classification is /cl FastForestClassification

After selecting Fast Forest from the learners drop list, the algorithm’s settings are displayed in the right pane of the TLC GUI.



#### Num Trees

Indicates the total number of trees constructed. The default value is 100. To change this value, enter the new value in the “Num Trees” box.

*On the Command Line:*

Add iter= followed by the number of trees to the learner settings string.

#### Num Leaves

Indicates the maximum number of leaves per tree. The default number is 20. To change this value, enter the new value in the “Num Leaves” box.

*On the Command Line:*

Add nl= followed by the number of leaves to the learner settings string.

#### Min Documents In Leaf

Indicates the minimum number of training instances required to form a leaf. The default number is 10. The number of instances is counted even if instances are weighted (not the total weight). To change this value, enter the new value in the “Min Instances In Leaf” box.

*On the Command Line:*

Add mil= followed by the value to the learner settings string.

#### Instance fraction

The fraction of instances (chosen randomly) to use for each tree. The default value is 0.7.

*On the Command Line:*

Add bagfrac= followed by the value to the learner settings string.

#### Feature fraction

The fraction of features (chosen randomly) to use for each tree. The default value is 0.7.

*On the Command Line:*

Add ff= followed by the value to the learner settings string.

#### Split fraction

The fraction of features (chosen randomly) to use on each split. The default value is 0.7.

*On the Command Line:*

Add sf= followed by the value to the learner settings string.

#### Quantile Enabled (regression only)

Indicates whether the quantile regression output is expected.

*On the Command Line:*

Add ge= followed by the value to the learner settings string.

#### Quantile sample count

If the quantile is enabled, then quantile sample count specifies the number of labels to be sampled in each leaf node.

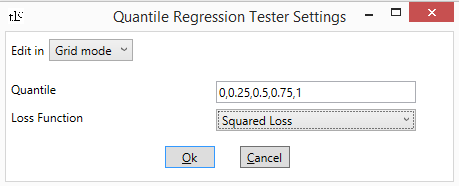
*On the Command Line:*

Add gsc= followed by the value to the learner settings string.

### Quantile Regression Tester

If Fast forest regression is the learner, the per instance output file will list out the [five point summary](http://en.wikipedia.org/wiki/Five-number_summary) for each instance. To compute different quantile values, “Quantile Regressor tester” should be chosen for evaluator as in the below picture

These are the options for Quantile Regression Tester



The Quantile values are comma separated values between 0 and 1 to list out in “per- instance” output file.

*On the Command Line:*

Use the command line argument /ev QuantileRegressorTester {quantiles=0,0.25,0.5} and specify the quantile values separated by commas.

## PCA Anomaly Detector

### Overview

This algorithm uses PCA to approximate the subspace containing the normal class. The subspace is spanned by orthonormal eigenvectors associated with the top eigenvalues of the data covariance matrix. More details on PCA can be read on [wikipedia](http://en.wikipedia.org/wiki/Principal_component_analysis).

For each new feature vector image002.png, the anomaly detector first computes its projection image004.png on the eigenvectors, then it computes the normalized norm error:

image005.png

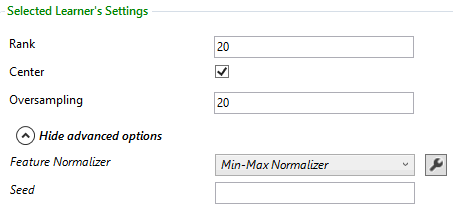
This norm error is the anomaly score. The higher the error, the more anomalous the instance is.

### Settings

*On the Command Line:*

The command-line argument for PCA Anomaly Detector is /cl pcaAnom.

After selecting PCA Anomaly Detection from the learners drop list, the algorithm’s settings are displayed in the right pane of the TLC GUI.



#### Rank

Indicates the number of components in the PCA. The default value is 20.

*On the Command Line:*

Add rank= followed by the value to the learner settings string.

#### Center

When this check box is checked,  the data is centered to have zero mean. It is checked by default.

*On the Command Line:*

Add center=- to the learner settings string to specify not centering the data.

#### Oversampling

This parameter is used in randomized PCA to control the accuracy. The default value is 20.

*On the Command Line:*

Add o= followed by the value to the learner settings string.

#### Feature Normalizer

By default, the features are normalized using the min-max normalizer. To specify no normalization, choose <None> from the Normalizer drop list. A different normalizer can also be chosen using this drop list.

*On the Command Line:*

* Add norm={} to the learner settings string to indicate no normalization
* Add norm=GaussianNormalizer or norm=BinNormalizer to the learner settings string to indicate normalization using a different normalizer. If a normalizer settings string is needed it follows the normalizer name, in a pair of curly brackets

#### Seed

The seed for the random number generator used by the trainer. If not specified, a random seed is used.

*On the Command Line:*

Add seed= followed by the value to the learner settings string.

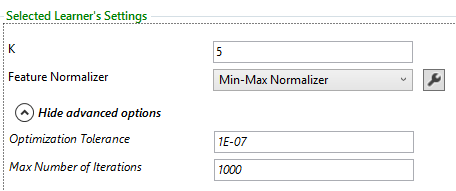
## K-means++

### Overview

[K-means](http://en.wikipedia.org/wiki/K-means_clustering) is a popular clustering algorithm. With K-means, the data is clustered into a specified number of clusters in order to minimize the within-cluster sum of squares. [K-means++](http://en.wikipedia.org/wiki/K-means%2b%2b) improves upon K-means by using a better method for choosing the initial cluster centers.

### Settings

After selecting Kmeans++ from the learners drop list, the algorithm’s settings are displayed in the right pane of the TLC GUI.



#### K

K is the number of clusters that will be found.

*On the Command Line:*

The command-line argument for K-means++ is /cl KmeansPlusPlus{k=n} where n is the number of clusters.

#### Optimization Tolerance

This parameter sets the threshold for the optimizer convergence. In other words, if the improvement between iterations is less than the threshold, the algorithm stops and returns the current model. The default value of this parameter is 1E-07.

*On the Command Line:*

Add ot= followed by the threshold to the learner settings string.

#### Feature Normalizer

By default, the features are normalized using the min-max normalizer. To specify no normalization, choose <None> from the Normalizer drop list. A different normalizer can also be chosen using this drop list.

*On the Command Line:*

* Add norm={} to the learner settings string to indicate no normalization
* Add norm=GaussianNormalizer or norm=BinNormalizer to the learner settings string to indicate normalization using a different normalizer. If a normalizer settings string is needed it follows the normalizer name, in a pair of curly brackets

#### Cluster Output

For each instance, k-means++ model outputs *k* values that are distances to the corresponding cluster centroids. The default output for /o option will only include 3 closest clusters (smallest 3 distances), but this can be changed by setting /ev=ClusteringTester{topk=<number of top clusters to output>}.

*On the Command Line:*

Use the command-line argument /o followed by the file name.

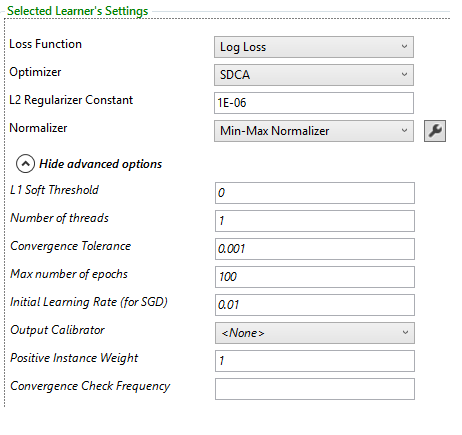
## Fast Linear Classifier

### Overview

The Fast Linear Classifier blends together the best of the [Logistic regression](#tlc_learners_logistic_regression_1181) and [SVM](#tlc_learners_svm_htm) algorithms. The new linear learner includes two state-of-the-art optimizers: SDCA (aka. [Stochastic Dual Coordinate Ascent](http://www.jmlr.org/papers/volume14/shalev-shwartz13a/shalev-shwartz13a.pdf)) and SGD ([Stochastic Gradient Descent](http://en.wikipedia.org/wiki/Stochastic_gradient_descent)). With this algorithm there is no need to specify a learning rate or number of iterations.  Native feature selection is performed with L1 regularization.

### Settings

After selecting Fast Linear Classifier from the learners drop list, the algorithm’s settings are displayed in the right pane of the TLC GUI.



#### Loss Function

The default loss is Log Loss. The other options are Hinge Loss and Smooth Hinged Loss.

*On the Command Line:*

Add lossFunction= followed by the name of the loss function and its options to the learner settings string.  Example

Available losses are LogLoss, HingeLoss and SmoothHingedLoss.

#### Optimizer

There are two choices for optimizing the predictor - SDCA or SGD.

*On the Command Line:*

Add o= followed by the name of the optimizer to the learner settings string.  Available optimizers are sdca and sgd.  SCDA is the default value.

#### L2 Regularizer Constant

The L2 [regularization](#ml_basics_regularization_htm) weight.

*On the Command Line:*

Add l2= followed by the weight to the learner settings string.  The default value is 1E-06.

#### Feature Normalizer

By default, the features are normalized using the min-max normalizer. To specify no normalization, choose <None> from the Normalizer drop list. A different normalizer can also be chosen using this drop list.

*On the Command Line:*

* Add norm={} to the learner settings string to indicate no normalization
* Add norm=GaussianNormalizer or norm=BinNormalizer to the learner settings string to indicate normalization using a different normalizer. If a normalizer settings string is needed it follows the normalizer name, in a pair of curly brackets

#### L1 Soft Threshold

L1 soft threshold (L1/L2). Note that it is easier to control and sweep using the threshold parameter than using the raw L1-regularizer constant.

*On the Command Line:*

Add l1threshold= followed by the float value of the threshold to the learner settings string.  The default value is 0.

#### Number of threads

Degree of lock-free parallelism.  A non-positive value means automatic, which depends on the sparseness of the data. Determinism isn't guaranteed.

*On the Command Line:*

Add threads= followed by the number of threads to the learner settings string.  The default value is 1.

#### Convergence Tolerance

Exponential moving averaged improvement tolerance for convergence.

*On the Command Line:*

Add convergenceTolerance= followed by the tolerance value to the learner settings string.  The default value is 0.001.

#### Max number of epochs

Maximum number of epochs.  Set to 1 to simulate online learning.

*On the Command Line:*

Add maxEpochs= followed by the number of epochs to the learner settings string.  The default value is 100.

#### Initial Learning Rate

The initial learning rate for SGD.

*On the Command Line:*

Add initLearningRate= followed by the initial value for the learning rate to the learner settings string.  The default value is 0.01.

#### Output Calibrator

Some classification algorithms produce raw scores that are unbounded. In order to convert these outputs into a binary prediction (or the predicted probability of belonging to one of the classes) the output must be calibrated. By default [sigmoid calibration](#introduction_classification_outp_9719). The other two options in the Calibrator drop list are [PAV calibration](#introduction_classification_outp_1316) and [Naïve calibrator](#introduction_classification_outp_6402).

*On the Command Line:*

* Add cali=PAVCalibration or cali=NaiveCalibration to the learner settings string to indicate calibration using a different calibrator
* Add cali={}to indicate no calibration

#### Positive Instance Weight

Apply weight to the positive class for imbalanced data.

*On the Command Line:*

Add piw= followed by the instance weight to the learner settings string.  The default value is 1.

#### ConvergenceCheckFrequency

Convergence check frequency in terms of the number of epochs.

*On the Command Line:*

Add checkFreq= followed by the check frequency to the learner settings string.  The default is the number of threads.

## Time Series Anomaly Detector

This algorithm uses untrained Ordinary Least Squares to detect anomalies in time series data. This simple algorithm works very well on typical telemetry data sets with minimal tuning. These algorithms can be used for ranking anomalies on batch data as well as online detection on streaming data.

The best way to use the new time-series anomaly detection functionality is through <http://PyTLC/>, which calls TL command under the hood and visualizes the results automatically. Here’s a sample template:  <http://pytlc/notebooks/TimeSeries.ipynb>.

### Sliding Window Data

The first step in time-series analysis is to convert time series data into sliding window data.  Sliding window data is a series of overlapping subsets of the time-series that can be used to identify anomalies in the series. To create sliding window data, use [CreateInstances](#other_tlc_modes_createinstances__1050) mode and add /writer=WindowWriter{options}where the options are:

* size= followed by the number data points to include in each window (e.g. 60 data points)
* stride= followed by the number of points to shift for each window (e.g. 20 data points)

Specify the output file with /cifile= followed by the filename.

Example:

*On the Command Line:*

TL /c CreateInstances Samples\TimeSeries\AppFailure.csv /inst=Text{sep=, name=0 nolabel=+} /writer=WindowWriter{size=60 stride=20} /cifile=windows.tsv

### Detect Anomalies

After creating the sliding window data set, you can score and rank anomalies with the [Test](#learning_modes_test_mode_htm) mode.

Example:

*On the Command Line:*

TL /c Test windows.tsv /inst=Text{name=0 nolabel=+} /pred=OLS

## Deep Structured Semantic Modeling

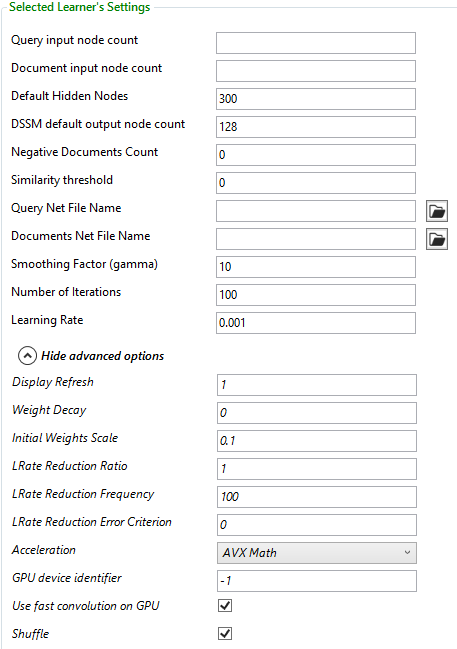
### Overview

DSSM is a Neural Network algorithm for learning feature embedding and cosine similarity for any key-value(s) pair such as: query and clicked through documents pairs, ads and documents pairs, entity-tweet pairs, or point and adjacent-points pairs. The basic idea can be found from [paper](http://research.microsoft.com/en-us/um/people/jfgao/paper/2013/cikm2013_DSSM_fullversion.pdf): “Learning Deep Structured Semantic Models for Web Search using Clickthrough Data”.

Currently, data in the form of key-values pairs can be consumed by DSSM. Each key has one positive (similar) value and a set of negative values. The number of negative values is configurable per dataset. DSSM run as multi-class predictor where the label is the index of the similar value within the set of all values. To call DSSM, key-value pairs need to be converted into instances.

### Settings

After selecting Dssm Neural Network from the multi-class learners drop list, the algorithm’s settings are displayed in the right pane of the TLC GUI.



#### Query input node count and Document input node count

These options are required, and their value must be equal to the number of features in the dataset.

*On the Command Line:*

* Add qfeats= followed by its value to the learner settings string.

#### Default Hidden Nodes

Setting this option overrides the default number of nodes in the hidden layers. By default, the network has two hidden layers, each with 300 neurons.

*On the Command Line:*

* Add qfeats= followed by the number of input nodes in the query net to the learner settings string
* Add dfeats= followed by the number of input nodes in the documents query net to the learner settings string

#### DSSM default output node count

Setting this option overrides the default number of nodes in the output layer of the query and the documents nets. The default number is 128.

*On the Command Line:*

Add outputsize= followed by the number of output nodes to the learner settings string.

#### Negative Documents count

This option sets the number of negative document (values) per training example.

*On the Command Line:*

Add negdocs= followed by the value to the learner settings string.

#### Similarity threshold

This is used only for reporting training errors after each training iteration. We are planning to show AUC instead. The default value is 0 for tanh activation function.

*On the Command Line:*

Add threshold= followed by the value to the learner settings string.

#### Query Net File Name and Documents Net File Name

These options allow defining the Net# files of the query (key) neural network and the documents (values) neural network. Nets with arbitrary architectures are supported with one restriction: the query and the documents neural networks must have the same number of the output nodes.

The default query and documents nets structure are two identical fully connected directed graph, containing one neuron for each feature in the input layer, 128 neurons in the output layer, and two hidden layers each consisting of 300 neurons. The default nets are used when no files are provided. If a net file is not specified, the number of nodes in the hidden layers can be changed by using the [Default Hidden Nodes](#tlc_learners_deep_structured_sem_6401) text box and the number of output nodes can be changed by using the [DSSM default output node count](#tlc_learners_deep_structured_sem_3126) text box.

*On the Command Line:*

* Add qryfilename= followed by name of the file containing the query net structure to the learner settings string
* Add docfilename= followed by the name of the file containing the documents net structure to the learner settings string

#### Smoothing factor (gamma)

A smoothing factor in the softmax function.

*On the Command Line:*

Add gamma= followed by its value to the learner settings string.

#### Other NN Settings supported:

Number of iterations, Learning Rate, Display Refresh, Weight Decay, Initial Weights Scale, Learning rate Reduction Ratio, Reduction Frequency, Reduction Error Criterion, Acceleration, Shuffle (see [Neural Networks settings](#tlc_learners_neural_networks_htm_6021) for details).

### How to use DSSM from the command line

Currently, DSSM accepts data in a specific format. Each training example should be an instance consisting of a query (key), a positive document (similar value), and optionally set of negative documents (negative values). Negative documents can be sampled from the positive documents of the other examples. The label of the instance is the index of the positive\similar value within the set of all values.

#### Example:

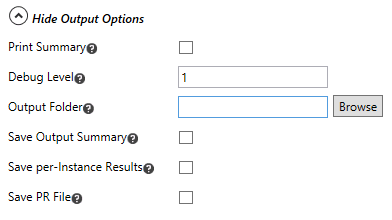
Let’s assume a dataset training file with a key and value columns suitable for learning similarity (e.g. Samples\AdSelection.pairs.txt). Follow the following steps to prepare the dataset for DSSM training:

1. Call TL.exe in [CreateInstances](#other_tlc_modes_createinstances__1050) mode to create the query instances, ignoring the document column. Any feature handler can be used. Rename the created file (same as the original input file name, with the additional suffix .tlc) to train.query.txt.tlc:  
   TL.exe /c CreateInstances "train.pair.txt" /inst TextInstances {label=0 attr=2 sep=tab handler=CharGram{cols=1}}  
   Note: If the noLabel=+ option is specified, the saved query instances are considered invalid instances by TL.exe and PrepareDssmData.exe. You can specify a column of dummy labels. The labels will get ignored and replaced with values indicating the index of the positive document within the set of documents generated by PrepareDssmData.exe.
2. Call TL.exe in [CreateInstances](#other_tlc_modes_createinstances__1050) mode to create the document instances, ignoring the query column. Rename the created file (same as the original input file name, with the additional suffix .tlc) to train.doc.txt.tlc:  
   TL.exe /c CreateInstances "train.pair.txt" /inst TextInstances {label=0 attr=1 sep=tab handler=CharGram{cols=2}}
3. Call PrepareDssmData.exe (standalone tool) with the following options:
   * /qfile option: allows specifying the query instances file (.tlc extension)
   * /dfile option: allows specifying the document instances file (.tlc extension)
   * /negdocs option (optional): allows specifying the number of negative documents per example required for DSSM training. The negative documents are sampled from the positive documents of the other examples.
   * /outfile option: allows specifying the output instances file where each instance contains a query, a positive document and -optionally- set of negative documents.  
     PrepareDssmData.exe /qfile="train.query.txt.tlc" /dfile="train.doc.txt.tlc" /outfile="train.pairs.txt" /negdocs=2  
     Note: The command line will print the query features count and document features count which are arguments to the DssmNet learner. The number of features count can be found also in train.query.txt.tlc and train.doc.txt.tlc files.
4. Call TL.exe to train DSSM on train.pairs.txt file:

TL.exe /c TrainTest /cl DssmNet{qfeats=18774 dfeats=17793 negdocs=2 iter=5 gamma=3 accel=sse} train.paris.txt /rs 1 /threads- /tlc /test train.paris.txt /o output.txt

# Choosing the output options

The output options are specified in the left pane of the TLC GUI, under “Advanced Options”.



## Print Summary

Checking this check box results in the [model summary](#output_model_formats_model_summa_3655) being printed to the standard output. It can be specified in all four learning modes, and is ignored in test mode. The output is different, depending on the learning algorithm that was used.

*On the Command Line:*

Use the command-line argument /summary+ to indicate printing the model summary to the standard output.

## Debug Level

Determines the detail with which the steps are displayed in the output. Can take values between 1 and 4. The default value is 1.

*On the Command Line:*

Use the command-line argument /debug followed by the value.

## Output folder

Specifies the directory in which the output is saved. If a folder is specified, the console output of every run is saved in a file named \*.out.txt, where \* is an automatically generated number for the algorithm that produced this output (when running in test mode, the file name is -1.out.txt).

The output folder is not to be confused with the [trained model folder](#learning_modes_train_mode_htm_tr_5991).

*On the Command Line:*

The full path of the file name to save must be specified. See [Saving the Model](#learning_modes_train_mode_htm_sa_1226) for details.

## Save Output Summary

Specifies that the output (the reported metrics) should be saved into a file. The file is located in the folder specified in the [Output folder](#choosing_the_output_options_htm__4388) text box, and is named \*.summary.txt, where \* is an automatically generated number for the algorithm that trained the model that produced these results (when running in test mode, the file name is .summary.txt).

*On the Command Line:*

Use the command-line argument /sf followed by the file name.

## Save per-Instance Results

Specifies that the per-instance results of the trained model should be saved into a file. The file is located in the folder specified in the [Output folder](#choosing_the_output_options_htm__4388) text box, and is named \*.inst.txt, where \* is an automatically generated number for the algorithm that trained the model (when running in test mode, the per-instance file name is .inst.txt). The file contains one line per instance, and for each instance the following numbers are reported: the true and predicted label (for classification) or value (for regression). For binary classifiers the uncalibrated result, the probability and the log-loss are also reported. For multi-class classifiers, the probability predicted for each class and the log-loss are also reported. For regressors, the L1 and L2 loss are also reported. For rankers, the numbers reported for each instance are the group key, the true score and the predicted score. In addition, NDCG@1,2,3, DCG@1,2,3 and MaxDCG@1,2,3 are reported (these are the same for all instances). When running in cross-validation mode, the file contains the per-instance results for each model trained (using the different folds).

*On the Command Line:*

Use the command-line argument /o followed by the file name.

**Note:** When using a weighted dataset with [FastTree](#tlc_learners_fasttree_htm) or [FastRank](#tlc_learners_boosted_trees_htm), the weight column is automatically included in the per-instance prediction file.

## Save PR File

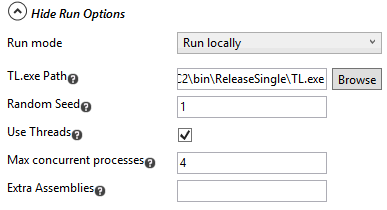
If this check box is checked, three files are generated. \*.pr.txt, containing the [recall and precision](#introduction_classification_meas_3157) data in text format, and \*.pr.pr.jpg, \*.pr.roc.jpg containing the PR and ROC curves (respectively). Here too, \* is an automatically generated number for the learner that produced these results (when running in test mode, these files are named .pr.txt, .pr.pr.jpg and .pr.roc.jpg respectively).

*On the Command Line:*

* Use the command-line argument /pr followed by the name of the text file
* The names of the other files are automatically generated by dropping the “.txt” suffix (if it exists) and adding “.pr.jpg” and “.roc.jpg” suffixes for the PR and ROC curves respectively.

# Choosing the run options

The RUN options are specified in the left pane of the TLC GUI, under “Advanced Options”.

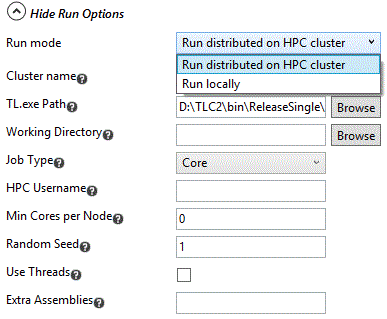


## Run mode

There are two options for run mode: Run locally or run distributed on HPC cluster.

### Run distributed on HPC cluster

Individual classifier runs can be parallelized on an HPC cluster.



Note that all the paths in the experiment (e.g. dataset and TLC binaries) need to be node-accessible, i.e. they can’t be on your local drive.

## TL.exe Path

Contains the path of the TL.exe executable that drives the run. It is auto-detected, and only needs to be changed for [HPC runs](#choosing_the_run_options_htm_run_3304).

## Random Seed

The seed of the pseudo random number generator. It is useful in order to reproduce duplicate runs.

*On the Command Line:*

Use the command-line argument /rs followed by seed to pass it to the program.

## Use Threads

To disable threading, uncheck the “Use Threads” check box. To enable it, check the “Use Threads” check box.

*On the Command Line:*

Use the command-line argument /threads- to disable threading.

## Max concurrent processes

The default value is equal to the number of cores on the machine. It can be changed to any number between 1 and the number of cores.

## Extra Assemblies

This text box is used to specify any extra assemblies the experiment needs to use (e.g, the [Strato Learner](#tlc_learners_stratolearner_htm) assembly).

*On the Command Line:*

Use the command line argument /dll followed by the full path of the assembly.

# Running Experiments

This section contains examples of running complete experiments using the GUI, including how to view and interpret the results. It also contains examples of running experiments using the command line.

## Running experiments on the command line

Following are examples of command-line runs that showcase some most commonly used options and algorithms. To run these as they are, copy [\\cloudmltlc\tlc\samples\](file://cloudmltlc/tlc/samples/) to the TL.exe directory.

*On the Command Line:*

TL.exe samples\breast-cancer.txt

Run averaged perceptrion, using 2-fold cross validation (default mode, default number of folds, default learner, data in default format).

*On the Command Line:*

TL.exe /c TrainTest samples\UCI\adult.train /test samples\UCI\adult.test /inst TextInstances { header=+ label=14 cat=1,3,5-9,13 sep=, }

Or:

TL.exe /c TrainTest samples\UCI\adult.train /test samples\UCI\adult.test @samples\UCI\adult\\_instset.txt

Run averaged perceptron. Train then test on a separate test file. Data is comma-separated, has a header, label in a specific column, and categorical features. If using the second option, the file \_instset.txt should contain the string/inst TextInstances { header=+ label=14 cat=1,3,5-9,13 sep=, } (other options can also be transferred to the file instead of appearing in the command-line).

*On the Command Line:*

TL.exe samples\SmartMatch\Instances-Relevance.txt /k 5 /inst TextInstances { header=+ name=0,1 } /strat /stratkey n0 /cl FR

Run the boosted tree classifier, using 5-fold cross-validation. Data has a header and name features in the first and second columns. Folds are stratified by the first name feature value.

*On the Command Line:*

TL.exe /cl MultiLR samples\UCI\iris\iris.txt /k 5 /summary+

Run multi-class Logistic Regression, using 5-fold cross-validation. Print the summary of the learned model for each fold.

*On the Command Line:*

TL.exe /cl LinearSVM { norm={} lambda=1e-4 } /c TrainTest samples\lm\Local.source\_features.de-de.txt /test samples\lm\Local.validate\_features.de-de.txt /inst TextInstances { header=+ text=3,4 lf=samples\lm\Mapping.de-de.txt label=0 attr=1,2 } /m model.bin /mc model.cs /mi model.ini /pr pr-curve.txt

Run linear SVM classifier using a separate file for training and for testing. Data has a header, two text features, and a text label. The mapping of the text label to binary labels is specified in the file samples\lm\Mapping.de-de.txt. Save the model in native binary, code, and INI formats. Save the PR curves to a file.

*On the Command Line:*

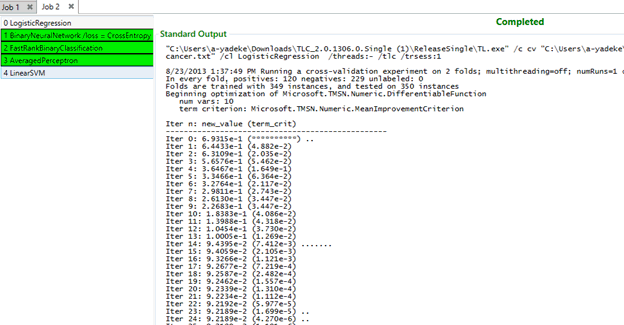
TL.exe samples\UnitTest\ranking-sample.txt /cl FRRank { iter=10 } /c TrainTest /test samples\UnitTest\ranking-sample.txt /inst TextInstances { header=+ name=1-2 groupKey=n0 lf=samples\UnitTest\ranking-sample.txt.labels } /m ranker.bin /mi ranker.ini

Run FastRank for ranking using 10 trees, with a separate file for training and test data. Data contains a header line, two name features and text labels. The grouping key is the first name feature. The mapping of the text labels to numeric is specified in the file samples\UnitTest\ranking-sample.txt.labels. Save the model in both binary format and ini format.

This dataset is in Bing Extraction format, so ExtractInstances identifies the label column, the query column, the grouping key and the label file automatically. The only thing that needs to be specified is the name in column number 2:

TL.exe samples\UnitTest\ranking-sample.txt /cl FRRank { iter=10 } /c TrainTest /test samples\UnitTest\ranking-sample.txt /inst ExtractInstances { name=2 } /m ranker.bin /mi ranker.ini

## Running experiments on TLC GUI



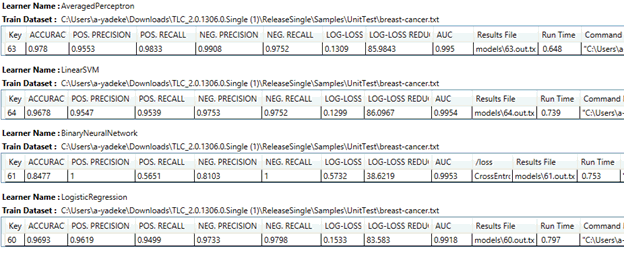
After choosing the algorithms and setting all the needed parameters, press F5, or click the “Run” button to run the experiment. As soon as it starts to run, TLC switches to the “Console Output” tab to show the immediate results. Learners highlighted in green, have completed their runs. An orange highlighting indicates that the learner produced a warning during its run. A blue highlighting indicates that the learner is running. A red highlighting indicates that the learner has failed to produce results.

For each run, TLC generates a new tab so that the results between different runs can be easily compared (assuming that settings have been tweaked for each run).

### Numerical results and comparison

Switch to the “Numerical Results” tab to see the numerical results and a comparison of different models trained by the different algorithms.

There, the results can be sorted by the metric of interest (such as accuracy or AUC).



### Keyboard shortcuts in TLC GUI

Keyboard shortcuts for the center pane of the GUI:

* Alt + L to focus on the learner dropdown list
* Alt + A to add the currently selected learner
* Keyboard shortcuts for parameter sweeps in the right pane of the GUI:
* Alt + G to create grid sweeps, or Alt + S to create random sweeps
* Alt + U to get suggested values for parameter sweeps
* Ctrl + T toggles the state of all the option lists between Show and Hide (lists that are expanded become collapsed and lists that are collapsed become expanded)

# ResultProcessor

Logs of previous runs can be processed using ResultProcessor.exe (this is how TLC GUI creates the “Results” tab).  In practice, this is **the way** manage your experiments and results!   Simply keep all your output files in some folders -- you can always get their summary by simply running:

ResultProcessor.exe [PathPattern]+ /o results.txt

for example:

ResultProcessor.exe c:\results\...\\*.out.txt /o results.txt

ResultProcessor.exe c:\results\1\\*.out.txt c:\new\...\\*.out.txt /o results.txt

as shown above, [PathPattern] can be a file or a regular expression, including '...' to go through all subdirectories.

In GUI, to store all \*.out.txt files, simply specify an Output Directory.

In command line, you can also run sweeps and summarize their results as follows:

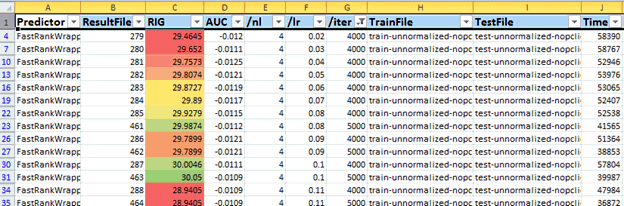
for %i in (1 10 100) do (TL.exe samples\breast-cancer.txt /k 10 /cls:iter:%i > %i.out.txt)ResultProcessor \*.out.txt /o results.txt

ResultProcessor has the following options:

|  |  |
| --- | --- |
| <resultFile patterns> | Result file patterns (can be multiple, can include '\*' and '...') |
| /outputFile=<string> | Output file name (short form /o) |
| /metrics=<string> | Which metrics should be processed (default=all)? (short form /a) |
| /includeStandardDeviations[+|-] | Include columns for standard deviations? Default value:'-' (short form /stdev) |
| /includePerFoldResults[+|-] | Output per-fold results Default value:'-' (short form /opf) |
| /perFoldResultSeparator=<string> | Separator for per-fold results.  Can be: actual char, 'tab', 'colon', 'space','comma' Default value:',' (short form /opfsep) |
| /extraAssemblies=<string> | Extra DLLs (short form /dll) |
| @<file> | Read response file for more options |

ResultProcessor creates tab-separated tables for each learner, where columns correspond to results for each parameter setting (auto-parsed).

The output of ResultProcessor can be copy-pasted into Excel, where you can get a single-click summary such as the following via Conditional Formatting/Color Scales (or make any plots if you wish from the data):



# TLC with SCOPE

In this release, TLC has started moving toward embedding common TLC Scope operations and define them as Scope functions in a module (MODULE/FUNC), instead of providing a bunch of Scope scripts as templates for users to modify in order to suit their machine-learning scenarios.

This Scope module (TlcLearner.module) and TLC binary will be released to a shareable Cosmos VC at http://cosmos08.osdinfra.net:88/cosmos/MSR.BigData/local/TLC/, so that TLC Scope users can use exactly the same officially released bits. In contrast, in the past, users usually submitted or ran their scripts referencing to some local TLC binary downloaded from a variety of TLC releases and/or private builds.

In this release, we did scalability tests against several TLC learners on Scope. This and several scripts came out of a benchmark effort for evaluating how TLC can scale (out) in a Big Data scenario as many TLC users have their datasets processed and stored on Cosmos.

We also have fixed several bugs related to handling categorical features in TLC Scope.

## TLC Learner Scope Module

The new TlcLearner.module has introduced two functions and also some experimental ones for testing purpose. We expect to add more functions in the future to suit a variety of Big Data training scenarios.

    FUNC Train

         RETURN  
         (  
             PartitionedSubModels ROWSET (PartitionIndex: string, PartitionedSubModel: byte[]),  
             TrainedModel ROWSET (CombinedModel: byte[])  
         )  
         PARAMS  
         (  
             TrainingInstanceRowset ROWSET(Instance: string),  
             modelTrainingArguments string,  
             combinerTrainingArguments string,  
             numberOfDataPartitions int  
         );

    FUNC Predict  
        RETURN  
        (  
            Predictions ROWSET (InstanceIndex: string, Label: string, RawPrediction: string, Probability: string,  
            Results ROWSET (ModelTag: string, Accuracy: string, PositivePrecision: string, PositiveRecall: string,  
                                 NegativePrecision: string, NegativeRecall: string, LogLoss: string, RIG: string, AUC: string),  
            PrecisionRecalls ROWSET (Recall: string, Precision: string, FalsePositiveRate: string, Output: string)  
        )  
        PARAMS  
        (  
            TestingInstanceRowset ROWSET(Instance: string),  
            trainedModelName string,  
            evaluationParsingArguments string,  
            modelTag string,  
            evaluatorArguments string  
        );

The Train function takes 4 parameters and returns two row-sets.

The parameters are:

* **TrainingInstanceRowset** – A text ROWSET used as the training set.
* **modelTrainingArguments** – Training arguments
* **combinerTrainingArguments** – Training arguments for an ensemble learner
* **numberOfDataPartitions** – Number of data partitions for training sub models

The returns of the Train function are two row-sets:

* **PartitionedSubModels** – A row-set of sub-models
* **TrainedModel** – The trained ensemble model, the main output from the Train function

The Predict function takes 5 parameters and returns 3 row-sets.

The parameters are:

* **TestingInstanceRowset** – A text ROWSET used as the testing set.
* **trainedModelName** – Name of the trained ensemble model. This model should be loaded as a RESOURCE.
* **evaluationParsingArguments** – Settings for evaluator parser
* **modelTag** – Model tag
* **evaluatorArguments** – Evaluator arguments

The returns of the Predict function are 3 row-sets:

* **Predictions** – Prediction outputs for the testing instances
* **Results** – Evaluation metrics
* **PrecisionRecalls** – Complete precision-recall values for all testing instances

## TLC Scope Module Scripts

In this release, we have included several scripts to demonstrate how to use the new TLC Scope module. You can find them in the Scope directory. Their purposes are self-explanatory as a training set is sharded into partitions and each trained for a sub-model of the specified learner.

* TlcLearnerModule-Train-ShardingLogisticRegression.script
* TlcLearnerModule-Train-ShardingBinaryNerualNetwork.script
* TlcLearnerModule-Train-ShardingFastTree.script
* TlcLearnerModule-Train-ShardingFastRank.script
* TlcLearnerModule-Predict-ShardingLogisticRegression.script
* TlcLearnerModule-Predict-ShardingBinaryNerualNetwork.script
* TlcLearnerModule-Predict-ShardingFastTree.script
* TlcLearnerModule-Predict-ShardingFastRank.script

We also have some data pre-processing scripts:

* TlcLearnerModule-Replicate.script
* TlcLearnerModule-Sample.script

## TLC Scope Module Deployment

As aforementioned, TLC module and TLC bits will be released to a Cosmos/Scope VC. We have asked MSR.BigData VC administrator to replicate and grant shareable access to this for any TLC users.

The official release will be located at

<http://cosmos08.osdinfra.net:88/cosmos/MSR.BigData/local/TLC/>

## TLC Scope Large-Scale Train and Prediction Scripts

These scripts are for internal uses. TLC users are encouraged to use TLC module functions as they are simpler and easier to read.

## Utility and Etc Scripts

In this release, we also included several utility Scope UDFs and scripts for better preparing datasets and some other purposes. However, they are mainly for internal uses only on some unique Big Data scenarios. Users are encouraged to use TLC module functions.

## Bug Fixes

In the past TLC Scope cannot handle categorical/text features as there is no easy way to collect all the categorical features and train a model in just one pass scanning through the dataset in a distributed environment. This bug has become quite an issue for TLC Scope users as many Cosmos datasets contain categorical or text features. The TLC team has fixed several bugs related to this issue and uses feature hashing to solve this problem once and for all.

# Aether: Learning via Aether modules

You can run TLC within Aether using the Aether TLC modules (search for non-official modules with the name “TLC”). Each module performs one of the primary functions of TLC:

|  |  |  |  |
| --- | --- | --- | --- |
| **Module** | **Function** | **Inputs** | **Outputs** |
| **TLC: Cross-Validation** | Cross validation | Training Data | Results |
| **TLC: Train then Test [with customDLL]** | Train and test a model. The “…with customDLL” allows you to attach a DLL that implements IPredictor in order to use TLC with your own learners. | Training Data  Test Data | Trained Model  Results |
| **TLC: Train** | Train a model | Training Data | Trained Model |
| **TLC: Test [with customDLL]** | Test a model. | Test Data  Trained Model | Results |
| **TLC: Standardize Instances [with test]** | Allows you to pre-normalize your data and/or reformat your data into standard TLC format. Use “…with Test” if you have train and test data to ensure the normalization is the same for both data sets. | Training Data | Standardized Training Data |
| **TLC: Feature Selection (CV)** | Perform feature selection using cross-validation for evaluation | Training Data | Feature Selection Output |
| **TLC: Feature Selection (Train-Test)** | Perform feature selection using training on a train set, testing on a test set, for evaluation. | Training Data | Feature Selection Output |

Effectively, each TLC module runs TL.exe. Thus, all of the usual command line parameters are available by double-clicking the module and setting its module parameters. To understand the various parameters for TLC, see [data formatting](#input_data_format_htm) and [predictor settings](#tlc_learners_htm) and/or run “TL.exe /?” to get a help listing.

To view how well the learner performed (when running cross-validation, test, or train-test), right-click the module and view its stdout (TLC reports things such as accuracy, precision, log-loss, etc.). Alternatively, you can download the results file which contains, for each tested instance, the true label, assigned label, and predictor output.

If you have normal [TLC text format](#input_data_format_text_input_for_6144) data, you can just plug it in to the Training Data or Test Data input pins to the TLC module. If you are using Bing extract data, you’ll need to change the instances reader to [Extract Instances](#input_data_format_bing_extractio_9537).

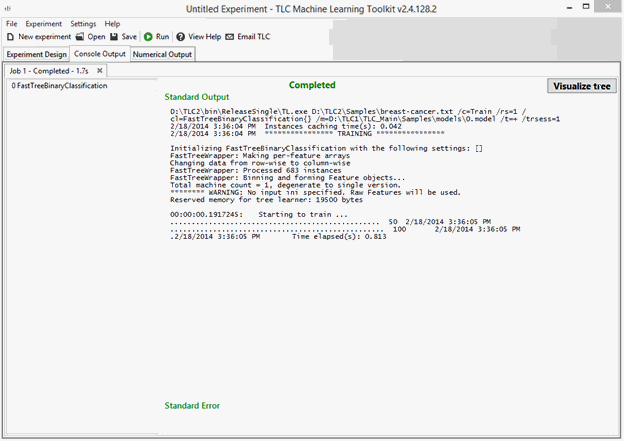
For a sample Aether experiment showing the use of all 7 modules, see [aether://experiments/c0b05db9-2652-4b45-b38f-35b576f7e8d6](file://D:\TLC3\TLC_Main\doc\OnlineDocumentation\aether:\\experiments\c0b05db9-2652-4b45-b38f-35b576f7e8d6).

# Understanding the model

## Tree Visualizer

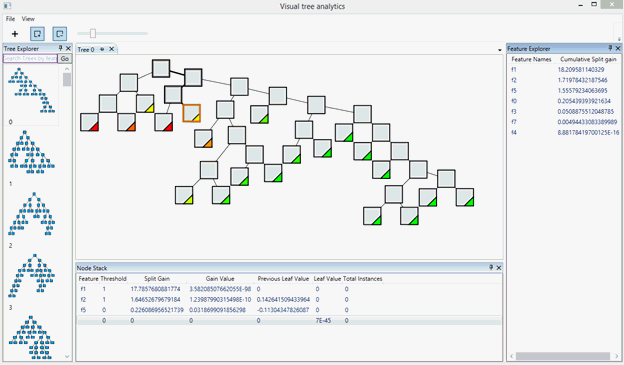
Tree Visualizer is a tool for visualizing decision trees created by binary, regression or ranking tree learners. It helps get the intuitive information about the trees, features and instances by visualizing it.

Tree Visualizer can be opened in three ways:

* Using the TLC GUI: click the "Visualize Tree" button in the top right corner of the console output Tab. This button is visible only for Fast tree Binary classifier, Fast tree Regression and Fast tree Ranking learners. If the model is not saved after training, an error message is shown when clicking "Visualize tree", as models files are necessary to invoke Tree Visualizer.  
  
* Using the command line: run TreeVisualizer.exe and load the model file and the dataset file by clicking the "Load Tree" button in the top left corner of the Tree Visualizer window (the dataset file is optional).
* Using the command line: run TreeVisualizer.exe and loading the model file and the dataset file using command line arguments.
  + /mf followed by the model file name (the model can be saved either in TLC format or ini format)
  + /ds followed by the dataset file name
  + /inst for the Instances settings. The syntax is the same as the [IPS string](#input_data_format_text_input_for_6144).
  + Example: /mf Models\0.model /ds Samples\UCI\adult.train/inst TextInstances{header=+ cat=1,3,5-9,13 sep=comma label=14}

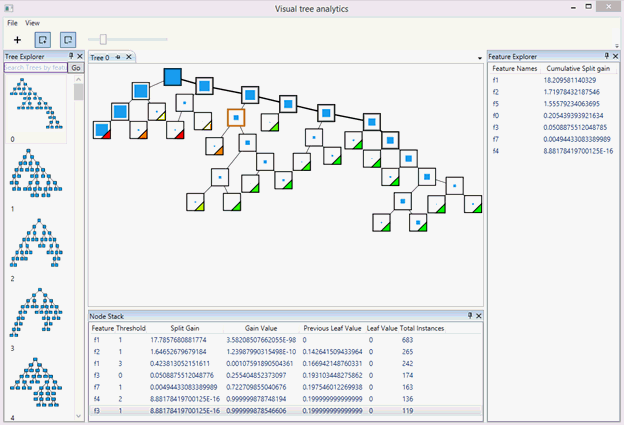
### Visualizing without data

The left pane of the window shows the Tree Explorer. The Tree Explorer contains thumbnails of the different trees in the model. The list of trees can be filtered to trees that contains a specific feature. Clicking on the thumbnail of a tree opens its visualization in a separate tab in the middle pane. The trees can be viewed in different layouts and zooms. The values in the leaves are encoded by color, going from red for negative values, to green for positive values. Double clicking on any node of the tree opens the Node Stack pane in the bottom of the window, which lists the features used to split from the root to the current node. The right pane contains the Feature Explorer, which lists the features used in the tree with their cumulative split gains. The Tree Explorer, the Node Stack and the Feature Explorer can be closed and reopened from the View menu.



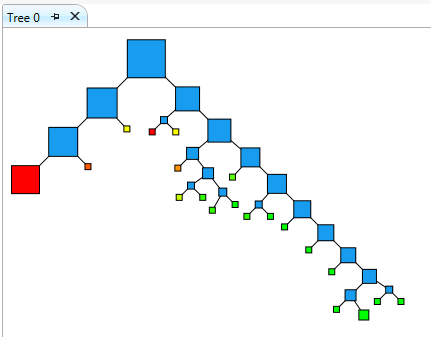
### Visualizing with data

When Tree Visualizer is loaded with a data file name, each node in the tree has a blue square in it. The area of the blue square represents the number of instances in the dataset that pass through the node. This number also appears in the Node Stack pane, and can be viewed as a tooltip by hovering with the mouse over a node.



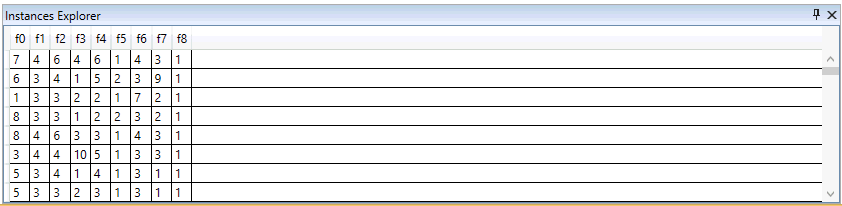
### Instances Only layout

By clicking the "Show instances only" icon (fourth icon from the left), the tree nodes are shown with size proportional to the instances flow on those nodes.



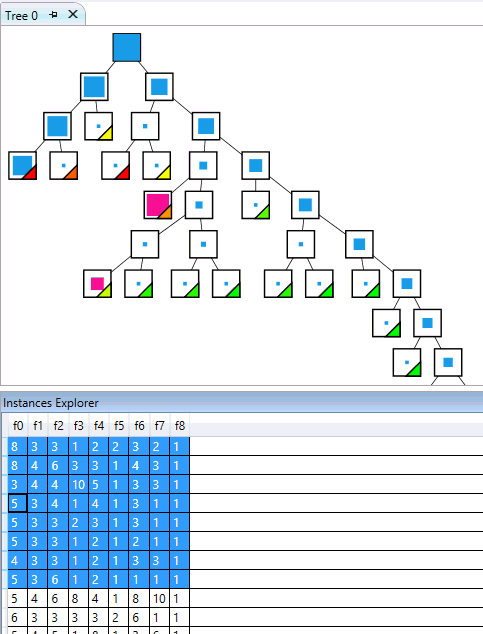
### Show instances on Grid

By right clicking a node and clicking show instances, all the instances are shown in the Instances Explorer pane.



### Selecting instances on Grid

When a subset of instances is selected, the nodes where these instances fall in are shown in pink. The size of the pink square is proportional to the number of selected instances in each node.



## ErrorExplorer: support for improving accuracy

ErrorExplorer (Rx) is primarily used for plugging in per-instance output files, which are obtained by specifying /o file.txt in TL.exe, or OutputInstanceResults in TLC (bottom-left corner).  Visualizations provided include:

* PR/ROC curves
* Output distribution density plots
* Per-class histograms of features/outputs
* Confusion matrix comparisons and support threshold adjustment
* GUI-based input slice selection from all of the above
* 2d and 3d visualization of any combinations of features and outputs
* Constraint-based slice selection (e.g., "select all positives misclassified with confidence>90% that have name containing "foo").

Error Explorer assumes your input data is same as the [default data format](#input_data_format_text_input_for_6144) for TLC. That is, your data should load into TLC without any need to specify instances settings. Data must be in dense format. Additionally, you cannot have any features named “Name” or “Label”, as ErrorExplorer treats these columns especially for the purposes of visualization. ErrorExplorer is designed to work for a small number of features (a few dozen, or low hundreds).

See <http://tlc/Home/ErrorExplorer> for screenshots, and second half of  [\\cloudmltlc\TLC\tlc\doc\TLC-Rx-demo.wmv](file:////cloudmltlc/tlc/doc/TLC-Rx-demo.wmv) for a workflow walkthrough.

# Net# (CloudML Neural Network Definition Language)

TLC includes a flexible neural network trainer/predictor. For details on how to use this predictor from within TLC, see the [Neural Networks](#tlc_learners_neural_networks_htm) section. This document covers the CloudML Neural Network Definition Language, known as Net# (formerly known as TNL). Net# is used to define a neural network structure to be trained by the TLC neural network trainer/predictor. Net# files use the .nn file extension. The TLC release includes several sample .nn files (typically paired with a .rsp file).

## Basic Concepts

A network structure, or net for short, consists of layers containing nodes, with weighted connections between nodes.

The connections are directional, each connection having a source node and destination node. Connections are organized into connection bundles, where all connections in a given bundle share the same source layer and the same destination layer. Net# views a connection bundle as belonging to the bundle’s destination layer.

There are three kinds of layers, input, hidden, and output. An input layer cannot be the destination layer of a bundle, and an output layer cannot be the source layer of a bundle. Conversely, a non-input layer must be the destination layer of at least one bundle, and a hidden layer must be the source layer of at least one bundle. If an input layer is not the source layer of at least one bundle, the neural network trainer issues a warning.

A Net# file specifies a network structure and optionally specifies initial values for the parameters of the net. Given a complete set of parameter values, the net can be used directly for prediction, without further training.

## Examples

This section contains three examples of varying complexity.

### Hello World Example

This is a simple net structure expressed in Net#:

input Data [100];  
hidden H [200] from Data all;  
output Out [10] from H all;

This network has an input layer, named Data, with 100 nodes, a hidden layer, named H, with 200 nodes that is fully connected to the input layer, and an output layer, named Out, with 10 nodes that is fully connected to the hidden layer.

An alternate encoding of this net is:

input Data auto;  
hidden H auto from Data all;  
output Out auto from H all;

When auto is used for the size of a layer, the Net# compiler attempts to determine the size automatically. Yet another encoding, that is equivalent to the one using auto is:

input Data param("InputNodeCount");  
hidden H param("HiddenNodeCount") from Data all;  
output Out param("OutputNodeCount") from H all;

See the sections on [auto](#net_sharp_user_guide_structure_s_7982) and [param](#net_sharp_user_guide_expressions_1605) for complete details.

### Sophomore Example

This is a more complex example:

// The input layers  
input Pixels [10, 20];  
input MetaData [7];

// Two hidden layers, based only on the pixels  
hidden ByRow [10, 12]  
  from Pixels where (s,d) => s[0] == d[0];  
hidden ByCol [5, 20]  
  from Pixels where (s,d) => abs(s[1] - d[1]) <= 1;

// A hidden layer based on both previous hidden layers  
hidden Gather [100] {  
  from ByRow all;  
  from ByCol all;  
}

// The output is based on the Gather hidden layer  
// and on the MetaData input layer  
output Result [10] {  
  from Gather all;  
  from MetaData all;  
}

This example illustrates several features:

         The structure has two input layers, Pixels and MetaData.

         The Pixels layer is a source for both ByRow and ByCol.

         The Gather and Result layers have multiple source layers.

         The ByRow and ByCol layers specify filtered connectivity using predicate expressions. More precisely, the node in ByRow at index [i, j] is connected to those nodes in Pixels having first index coordinate equal to the node’s first coordinate, i. Similarly, the node in ByCol at [i, j] is connected to those nodes in Pixels having second index coordinate within one of the node’s second coordinate, j.

### Digit Recognition Example

Here is a convolutional network intended to recognize numeric digits, to be trained from the MNIST data set, which consists of 28 x 28 gray scale images.

input Image [28, 28];

hidden Conv1 [5, 13, 13]  
  from Image convolve {  
    InputShape  = [28, 28];  
    UpperPad    = [ 1,  1];  
    KernelShape = [ 5,  5];  
    Stride      = [ 2,  2];  
    MapCount    = 5;  
  }

hidden Conv2 [50, 5, 5]  
  from Conv1 convolve {  
    InputShape  = [ 5, 13, 13];  
    KernelShape = [ 1,  5,  5];  
    Stride      = [ 1,  2,  2];  
    Sharing     = [false, true, true];  
    MapCount    = 10;  
  }

hidden Hid3 [100] from Conv2 all;

output Digit [10] from Hid3 all;

When paired with suitable options, TLC will train this net to about 99% accuracy. The TLC samples include MNIST\Conv28.nn and MNIST\Conv28.rsp that demonstrate this. The TLC neural network trainer includes support for hardware acceleration (SSE, AVX, GPU, MKL). With SSE, MNIST\Conv28.rsp trains in about 5 minutes. Note that the AVX instruction set is only supported by newer processors. Also note that early implementations of AVX are less efficient than ideal—the expected 2x improvement over SSE is only achieved with the new Haswell chips.

## Layers

As illustrated in the examples, Net# supports three kinds of layers, input, hidden, and output. Input layers have no associated trained parameters and represent entry of instance data into the network. Trainable layers, namely hidden and output layers, have associated trained parameters, known as weights and biases.

A layer has a fixed number of nodes, conceptually arranged as a rectangular array of arbitrary dimension. In the digit recognition example, the trainable layers Hid3 and Digit each have a one-dimensional arrangement of nodes, while the other layers use two or three-dimensional arrangements.

A trainable layer has one or more connection bundles. In the sophomore example, the Gather and Result layers each have two connection bundles. A connection bundle consists of a source layer and a specification of the connections to that source layer. Net# currently supports three kinds of connection bundles, namely, full bundles, indicated by the keyword all, filtered bundles, indicated by the keyword where followed by a predicate expression, and convolutional bundles, indicated by the keyword convolve followed by the convolution properties.

Note that the connection bundles define both a coarse-grained directed graph structure on the layers of a network, as well as a fine-grained directed graph structure on the nodes of the network.

Currently, the TLC neural network predictor imposes the following restrictions:

         There must be exactly one output layer.

         An output layer cannot be the source layer for a connection bundle.

         The directed graph of layers must be acyclic.

## Type System

Net# supports the primitive types, { Integer, Float, Boolean, String }. Integers are 32 bits. Floating point values are either Single or Double precision, depending on the build of TLC.

Additionally, Net# supports homogeneous tuples of these types. The length of a tuple is part of its type. Moreover, a primitive type is considered equivalent to a tuple type of length one. Consequently, a Net# type is really a pair consisting of a primitive type together with a positive integer (Net# does not support empty tuples of length zero).

Since a primitive type is considered equivalent to a tuple type of length one, whenever we specify that a value is a tuple, we include the case of a primitive value.

Net# does not support the notion of null.

## Values and Constants

Net# supports symbolic constants. For example, the digit recognition example could be specified as:

// "Magic numbers" are here.  
const {  
  ImageSize = 29; // Padded size  
  Pad = 1;  
  KernSize = 5;  
  Step = 2;  
  Maps1 = 5;  
  Maps2 = 10;  
}

// These are derived constants.  
const {  
  ImageDims = [ImageSize – Pad, ImageSize - Pad];  
  KernDims = [KernSize, KernSize];

  ConvSize1 = (ImageSize - KernSize) / Step + 1;  
  ConvDims1 = [Maps1, ConvSize1, ConvSize1];

  ConvSize2 = (ConvSize1 - KernSize) / Step + 1;  
  ConvDims2 = [Maps2 \* Maps1, ConvSize2, ConvSize2];  
}

// Aliases for readability.  
const { F = false; T = true; }

// The layers  
input Image ImageDims;

hidden Conv1 Maps1 \* ConvSize1 \* ConvSize1  
  from Image convolve {  
    InputShape  = ImageDims;  
    UpperPad    = [Pad, Pad];  
    KernelShape = KernDims;  
    Stride      = [Step, Step];  
    MapCount    = Maps1;  
  }

hidden Conv2 [ConvDims2]  
  from Conv1 convolve {  
    InputShape  = ConvDims1;  
    KernelShape = [1] & KernDims; // Concatenation  
    Stride      = [1, Step, Step];  
    Sharing     = [F, T, T];  
    MapCount    = Maps2;  
  }

hidden Hid3 [100] from Conv2 all;

output Digit [10] from Hid3 all;

This illustrates several features of Net# :

         Net# supports standard operators on primitive types, as well as some tuple operators, such as the concatenation operator A & B.

         The standard Boolean values are false and true. This script defines F and T to be aliases for these values.

         Net# flattens tuples, so [[1,2,3],4,[5,6]] is equivalent to [1,2,3,4,5,6]. For example, the dimensions of Conv2 is specified as [ConvDims2], but could have been specified simply as ConvDims2, since both reduce to [50,5,5]. The extra brackets are optional and are used to enhance readability. Similarly, the size of the Digit layer was written [10], but could have been 10, or 2 \* 5. Also, note that the KernelShape property in Conv2 could have been specified as [1, KernDims] instead of using the concatenation operator, [1] & KernDims.

         Note that the dimensionality of Conv1 is specified as an Integer value. Nevertheless, the InputShape for the convolutional bundle in Conv2 is a tuple of length three. This is legitimate as long as the total numbers of nodes agree. That is, the products of the dimensions must match.

## Default Structure

The TLC neural network predictor provides a default network structure when no Net# file is provided. The default structure is of the form:

input Data auto;

hidden H auto from Data all;

output Result auto from H all;

## Language Basics

Net# is a declarative language. Generally, elements can be specified in any order. For example, constant declarations can come after the constants are used, the layers can be declared in any order, etc. One place where order does matter is when there are multiple input layers. The declared order of the input layers should match the order of features in the input data.

Net#’s lexical grammar and rules are very similar to those of C#. For example:

         Net# is case sensitive.

         Net# supports standard C# style comments of both kinds.

         Net# constant literals are similar to C#, including decimal and hexadecimal integer literals, floating point literals, and string literals (including verbatim string literals) with escape sequence support.

         Prefixing a keyword with the @ character makes it a normal identifier.

The keywords of Net# are:

true  
false  
not  
and  
or

const  
input  
output  
hidden  
share

from  
all  
where  
convolve  
pool  
norm  
auto

Net# also has some contextual keywords, that have special meaning in certain contexts, including:

sigmoid  
linear  
softmax  
rlinear  
square  
sqrt  
srlinear  
tanh  
brlinear

abs  
bittest  
param

max  
mean  
response

Net# expressions are strongly typed using type inference; the Net# code does not need to (and indeed cannot) explicitly state the type of an expression.

## Structure Specification

A neural network structure specification expressed in Net# must conform to the syntactic grammar detailed in this and the next section. This page focuses on the structural elements. The [next page](#net_sharp_user_guide_expressions_8982) focuses on expressions.

### Net

net:  
sections

sections:  
section  
sections    section

section:  
const-section­  
input-section  
hidden-section  
output-section  
share-declaration

A Net# file consists of sections and share-declarations. A share-declaration specifies that certain weights and/or biases be shared.

There are four kinds of sections, const, input, hidden, and output. Each section kind has a corresponding declaration kind. The const sections define symbolic constants, while the other sections define layers in the network structure.

const-section:  
const    const-declaration  
const    {    const-declarationsopt    }

input-section:  
input    input-layer-declaration  
input    {    input-layer-declarationsopt    }

hidden-sections:  
hidden    trainable-layer-declaration  
hidden    {    trainable-layer-declarationsopt    }

ouput-section:  
output    trainable-layer-declaration  
output    {    trainable-layer-declarationsopt    }

A section consists of the appropriate keyword followed by either a single declaration or any number of declarations enclosed in curly braces. For example:

const X = 17 \* 2;

const { Y = 3; Z = true; }

input Data [10];

input { Pixel [10, 10]; MetaData [17]; }

### Constant Declarations

const-declarations:  
const-declaration  
const-declarations    const-declaration

const-declaration:  
identifier    =    constant-value    ;

constant-value:  
expression

A constant declaration consists of an identifier followed by an equal sign and a constant value expression. The constant expression must reduce to a value of a known type.

### Layer Declarations

input-layer-declarations:  
input-layer-declaration  
input-layer-declarations    input-layer-declaration

input-layer-declaration:  
name    dimensions    ;

trainable-layer-declarations:  
trainable-layer-declaration  
trainable-layer-declarations    trainable-layer-declaration

trainable-layer-declaration:  
name    dimensions    output-functionopt    connection-bundle-and-property-set

output-function:  
sigmoid  
linear  
softmax  
rlinear  
square  
sqrt  
srlinear  
abs  
tanh  
brlinear

name:  
identifier

dimensions:  
expression  
auto

A layer declaration defines a layer, including its connection bundles and properties. Each layer declaration starts with the name of the layer, followed by the dimensions of the layer. The dimensions consists of either a tuple of positive integers or the keyword auto. In the former case, the product of those integers is the number of nodes in the layer.

When dimensions is auto, the Net# compiler determines the number of nodes automatically. For input layers, this is determined by inspecting the training data. For trainable layers (hidden and output), the auto size is the natural size of the first connection bundle belonging to the layer. The natural size of a convolutional, pooling, or response normalization bundle is determined by its geometry. In a hidden layer, the natural size of a full or filtered bundle is the value of the defaultHiddenNodes hyper-parameter. In an output layer, the natural size of a full or filtered bundle is one for regression, two for binary classification, and the value of the defaultOutputNodes hyper-parameter for multi-class classification.

A trainable layer declaration (output or hidden layer) can optionally include the output-function, which typically defaults to sigmoid. When all bundles of a layer are pooling or response norm bundles, the layer's output-function defaults to linear.

connection-bundle-and-property-set:  
bundle-spec  
{    bundle-specs-and-properties    }

bundle-specs-and-properties:  
bundle-spec  
bundle-specs-and-properties    bundle-spec  
property  
bundle-specs-and-properties    property

The connection-bundle-and-property-set of a trainable layer consists of either a single bundle-spec or a set of bundle-specs and properties enclosed in curly braces.

bundle-spec:  
identifier    all    ;  
identifier    all    property-set  
identifier    where    (    identifier    ,    identifier    )    =>    expression    ;  
identifier    where    (    identifier    ,    identifier    )    =>    expression    property-set  
identifier    convolve    property-set  
identifier    meanopt    pool     property-set  
identifier    max    pool    property-set  
identifier    response    norm    property-set

Net# currently supports five kinds of connection bundles, namely full (indicated by all), filtered (indicated by where), convolutional (indicated by convolve), pooling (indicated by max pool or mean pool) and response normalization (indicated by response norm).

Each connection bundle may have properties. Note that properties are required for convolutional, pooling and response normalization bundles, but optional for the other bundles.

property-set:  
{    properties    }

properties:  
property  
properties    property

property:  
name    =    value    ;

value:  
expression

Note that a property name is not a keyword. However, only pre-defined properties are legal.

#### Properties for Trainable Layers

For trainable layers (hidden and output), the only property currently defined is Biases, and is optional. If specified, the layer must support trainable biases and the value of the property must be a tuple of floating point values whose length matches the number of nodes in the layer. The default initial bias values are randomly generated.

A layer supports trainable biases when at least one of its bundles requires biases. Convolutional bundles have integrated biases (that are shared), so do not require biases on the layer. Similarly, pooling and response normalization bundles do not require biases. In contrast, full and filtered connection bundles require biases on the layer.

#### Full Bundle

A full bundle includes a connection from each node in the source layer to each node in the destination layer.

The only property currently defined for a full bundle is Weights, and is optional. The value must be a tuple of floating point values whose length matches the number of connections defined by the bundle. The number of connections is the number of nodes in the source layer times the number of nodes in the destination layer. The default weights are randomly generated.

Weight values are grouped by the destination node index. That is, if the source layer has N nodes and the destination layer has M nodes, then the Weights tuple should have M \* N elements, where the first N elements are the weights for the first destination node, the next N elements are the weights for the second destination node, etc.

#### Filtered Bundle

A filtered bundle specification includes a predicate, expressed syntactically much like a C# lambda expression.

The sophomore example includes two filtered bundles:

input Pixels [10, 20];

hidden ByRow[10, 12]  
  from Pixels where (s,d) => s[0] == d[0];  
hidden ByCol[5, 20]  
  from Pixels where (s,d) => abs(s[1] - d[1]) <= 1;

In the predicate for ByRow, s is a parameter representing an index into the rectangular array of nodes of the input layer Pixels, while d is a parameter representing an index into the array of nodes of the hidden layer ByRow. The type of both s and d is a tuple of integers of length two. Conceptually, s ranges over all pairs of integers with 0 <= s[0] < 10 and 0 <= s[1] < 20, while d ranges over all pairs of integers with 0 <= d[0] < 10 and 0 <= d[1] < 12. The predicate expression indicates that the bundle includes a connection from the node defined by s to the node defined by d iff s[0] is equal to d[0].

The only property currently defined for a filtered bundle is Weights, and is optional. The value must be a tuple of floating point values whose length matches the number of connections defined by the bundle. The default weights are randomly generated.

Weight values are grouped by the destination node index. That is, if the first destination node is connected to K source nodes, then the first K elements of the Weights tuple are the weights for the first destination node, in source index order. Similarly for the remaining destination nodes.

#### Convolutional Bundle

When the training data has homogeneous structure, e.g., spatial or temporal dimensionality, as in image, audio, or video data, convolutional connections are commonly used to learn high level features of the data. Convolutional bundles employ rectangular kernels that are slid through the dimensions. Essentially, each kernel defines a set of weights applied in local neighborhoods, referred to as kernel applications. Each kernel application corresponds to a node in the source layer referred to as the central node. The weights of a kernel are shared among many connections. In a convolutional bundle, each kernel is rectangular and all kernel applications are the same size.

Here are some sites, suggested by Bing, that discuss convolutional networks:

* <http://deeplearning.net/tutorial/lenet.html>
* <http://research.microsoft.com/pubs/68920/icdar03.pdf>
* <http://people.csail.mit.edu/jvb/papers/cnn_tutorial.pdf>

To learn more, feel free to ask for assistance on [tlcdisc](mailto:tlcdisc).

Convolutional bundles support the following properties:

         InputShape: This required property defines the dimensionality of the source layer for the purposes of this convolutional bundle. The value must be a tuple of positive integers. The product of the integers must equal the number of nodes in the source layer, but otherwise does not need to match the dimensionality declared for the source layer. The length of this tuple is called the arity of the convolutional bundle.

         KernelShape, Stride, Padding, LowerPad and UpperPad define the shape and locations of the kernels.

         KernelShape: This required property defines the dimensionality of each kernel for the convolutional bundle. The value must be a tuple of positive integers whose length is the arity of the bundle. Each component of this tuple must be no greater than the corresponding component of the InputShape.

         Stride: This optional property defines the sliding step sizes of the convolution (one step size for each dimension), i.e. the distance between the central nodes. The value must be a tuple of positive integers whose length is the arity of the bundle. Each component of this tuple must be no greater than the corresponding component of the KernelShape. The default value is a tuple with all components equal to one.

         Padding: This optional property determines whether the input should be padded using a default padding scheme. The value may be either a single Boolean value or a tuple of Boolean values whose length is the arity of the bundle. A single Boolean value is extended to be a tuple of the correct length with all components equal to the specified value. If the value for a dimension is true, the source is logically padded in that dimension with zero-valued cells to support additional kernel applications, such that the central nodes of the first and last kernels in that dimension are the first and last nodes in that dimension in the source layer. Thus, the number of “dummy” nodes in each dimension is determined automatically, to fit exactly (InputShape[d] – 1) / Stride[d] + 1 kernel applications in the padded source layer. If the value for a dimension is false, the central nodes are chosen so that the number of nodes on each side that are left out is the same (up to a difference of 1). The default value of this property is a tuple with all components equal to false.

         UpperPad and LowerPad: These two optional properties provide fine control over the amount of padding to use. They can be defined only if Padding is not defined. They are integer valued tuples whose lengths are the arity of the bundle. When these properties are specified, “dummy” nodes are added to the lower and upper ends of each dimension of the input layer. The number of nodes added to the lower and upper ends in dimension i is determined by LowerPad[i] and UpperPad[i], respectively. To ensure that central nodes are "real" nodes and not "dummy" nodes, each component of LowerPad must be strictly less than KernelShape[d]/2 and each component of UpperPad must be no greater than KernelShape[d]/2. The default value of these properties is a tuple with all components equal to 0.

         Sharing: This optional property defines the weight sharing for each dimension of the convolution. The value may be either a single Boolean value or a tuple of Boolean values whose length is the arity of the bundle. A single Boolean value is extended to be a tuple of the correct length with all components equal to the specified value. The default value is a tuple consisting of all true values.

         MapCount: This optional property defines the number of feature maps for the convolutional bundle. The value may be either a single positive integer or a tuple of positive integers whose length is the arity of the bundle. A single integer value is extended to be a tuple of the correct length with the first components equal to the specified value and all the remaining components equal to one. The default value is one. The total number of feature maps is the product of the components of the tuple. The factoring of this total number across the components determines how the feature map values are grouped in the destination nodes.

         Weights: This optional property defines the initial weights for the bundle. The value must be a tuple of floating point values whose length is the number of kernels times the number of weights per kernel, as defined below. The default weights are randomly generated.

The number of weights per kernel is one plus the product of the components of KernelShape. The extra one acts as an integrated (shared) bias value.

The center of the kernel is defined as the location in the kernel with coordinates (KernelShape[d]-1)/2, where / denotes integer division. Each kernel application aligns the center of the kernel with the central node (within the source) of the application.

The InputShape, KernelShape, Stride, Padding, LowerPad and UpperPad determine the number of nodes produced by each feature map as the product of NodeCount[d] across 0 <= d < arity. In general NodeCount[d] is computed as:

NodeCount[d]=1+(Width[d]–KernelShape[d])/Stride[d]

Where Width[d] depends on the padding options. In particular, when there is no padding:

Width[d]=InputShape[d]

When Padding[d] is true:

Width[d]=InputShape[d]+KernelShape[d]-1

And when LowerPad and UpperPad are specified instead of Padding:

Width[d]=InputShape[d]+LowerPad[d]+UpperPad[d]

Recall that the division operator in the preceding formula for NodeCount is integer division; generally, the numerator does not need to be divisible by Stride[d]. However, when LowerPad[d] or UpperPad[d] is positive, the numerator is required to be divisible by Stride[d], so the kernel applications fit exactly into the corresponding padded source layer dimension. I.e., with positive LowerPad or UpperPad, there cannot be dummy nodes that do not participate in any kernel applications.

The number of nodes per feature map times the product of the components of MapCount must equal the number of nodes declared for the destination layer. If the destination layer is declared with auto size (and this bundle is its first), then this computation defines the number of nodes in the layer.

When the components of Sharing are all true, the number of kernels is the product of the components of MapCount. Otherwise, the number of kernels is that product times the product of NodeCount[d] as d ranges over dimensions for which Sharing[d] is false.

The number of weights is the number of kernels times the number of weights per kernel.

As an example, consider the convolutional bundle for Conv1 in the digit recognition example:

input Image [28, 28];

hidden Conv1 [5, 13, 13]  
  from Image convolve {  
    InputShape  = [28, 28];  
    UpperPad    = [ 1,  1];  
    KernelShape = [ 5,  5];  
    Stride      = [ 2,  2];  
    MapCount    = 5;  
  }

         The arity of the convolution is two (the length of the tuples InputShape, UpperPad, KernelShape, and Stride).

         The number of weights per kernel is 1 + KernelShape[0] \* KernelShape[1] = 1 + 5 \* 5 = 26.

         NodeCount[0] = (28 + 1 - 5) / 2 + 1 = 13.

         NodeCount[1] = (28 + 1 – 5) / 2 + 1 = 13.

         The total number of nodes is MapCount \* NodeCount[0] \* NodeCount[1] = 5 \* 13 \* 13, matching the declared dimensionality of the layer, [5, 13, 13].

         Since Sharing defaults to true, the number of kernels is the same as the number of maps, namely 5, and the number of weights is 26 \* 5 = 130.

An alternative to using UpperPad is to transform the training data so it consists of 29 x 29 images. Then the Net# would be:

input Image [29, 29];

hidden Conv1 [5, 13, 13]  
  from Image convolve {  
    InputShape  = [29, 29];  
    KernelShape = [ 5,  5];  
    Stride      = [ 2,  2];  
    MapCount    = 5;  
  }

The computations for Conv2 are similar but also involve Sharing:

input Image [28, 28];

hidden Conv1 [5, 13, 13]  
  from Image convolve {  
    InputShape  = [28, 28];  
    UpperPad    = [ 1,  1];  
    KernelShape = [ 5,  5];  
    Stride      = [ 2,  2];  
    MapCount    = 5;  
  }

hidden Conv2 [50, 5, 5]  
  from Conv1 convolve {  
    InputShape  = [ 5, 13, 13];  
    KernelShape = [ 1,  5,  5];  
    Stride      = [ 1,  2,  2];  
    Sharing     = [false, true, true];  
    MapCount    = 10;  
  }

         The arity of the convolution is three (the length of the tuples InputShape, KernelShape, Stride and Sharing).

         The number of weights per kernel is 1 + KernelShape[0] \* KernelShape[1] \* KernelShape[2] = 1 + 1 \* 5 \* 5 = 26.

         NodeCount[0] = (5 - 1) / 1 + 1 = 5.

         NodeCount[1] = (13 - 5) / 2 + 1 = 5.

         NodeCount[2] = (13 – 5) / 2 + 1 = 5.

         The total number of nodes is MapCount \* NodeCount[0] \* NodeCount[1] \* NodeCount[2] = 10 \* 5 \* 5 \* 5, matching the declared dimensionality of the layer, [50, 5, 5].

         Since Sharing[d] is false only for d == 0, the number of kernels is MapCount \* NodeCount[0] = 10 \* 5 = 50.

         The number of weights is 26 \* 50 = 1300.

#### Pooling Bundle

Pooling bundles apply geometry similar to convolutional bundles, but apply predefined functions to source node values to determine a destination node value. Hence, pooling bundles have no trainable state (weights or biases). Pooling bundles support all the convolutional properties except Sharing, MapCount, and Weights.

Traditionally, the kernels summarized by adjacent pooling units do not overlap. If Stride[d] is equal to KernelShape[d] in each dimension, the layer obtained is the traditional local pooling layer, commonly employed in convolutional neural networks. Each destination node computes either the maximum or the mean of the activities of its kernel application in the source layer.

Here are some references that discuss pooling layers:

* Section 3.4 in: <http://www.cs.toronto.edu/~hinton/absps/imagenet.pdf>
* <http://cs.nyu.edu/~koray/publis/lecun-iscas-10.pdf>
* <http://cs.nyu.edu/~koray/publis/jarrett-iccv-09.pdf>

As an example, consider the following pooling bundle:

hidden P1 [5, 12, 12]  
  from C1 max pool {  
    InputShape  = [ 5, 24, 24];  
    KernelShape = [ 1,  2,  2];  
    Stride      = [ 1,  2,  2];  
  }

         The arity of the bundle is three (the length of the tuples InputShape, KernelShape and Stride).

         The number of nodes in the source layer is 5 \* 24 \* 24 = 2880.

         This is a traditional local pooling layer, since KernelShape and Stride are equal.

         The number of nodes in the destination layer is 5 \* 12 \* 12 = 1440.

#### Response Normalization Bundle

Response normalization is a scheme that was first introduced in section 3.3 of [this ImageNet paper](http://www.cs.toronto.edu/~hinton/absps/imagenet.pdf). It is used to aid generalization in neural nets. When one neuron is firing at a very high activation level, a local response normalization layer suppresses the activation level of the surrounding neurons. This is done by using three parameters α, β and k, and a convolutional structure (or neighborhood shape). Every neuron y in the destination layer corresponds to a neuron x in the source layer. The activation level of y is given by

image002.png

where f is the activation level of a neuron, and Nx is a set containing the neurons in the neighborhood of x defined by the convolutional structure (referred to as a kernel). If the kernel contains neurons in the same map as x, the normalization is referred to as same map normalization, whereas if the kernel contains neurons in the same spatial position as x, but in other maps, it is referred to as across maps normalization. This sort of response normalization implements a form of lateral inhibition inspired by the type found in real neurons, creating competition for big activation levels amongst neuron outputs computed on different maps.

Since response normalization bundles apply a predefined function to source node values to determine the destination node value, they have no trainable state (weights or biases). Response normalization bundles support all the convolutional properties except for Sharing, MapCount, and Weights. To define same map normalization, the first coordinate in InputShape must have the value 1 and to define across maps normalization, the first coordinate must be an integer greater than one and no greater than the number of maps, and the rest of the coordinates must have the value 1.

Note that the nodes in the destination layer correspond to neurons x, which are the central nodes of the kernels (i.e. at (KernelShape[d]-1)/2 where / denotes integer division). Therefore, if Padding[d] is false this means that the first and the last KernelShape[d]/2 nodes do not have corresponding nodes in the destination layer. To avoid this situation, set Padding to true.

In addition to the four properties described above, response normalization bundles also support the following properties:

         Alpha: This required property is a floating-point parameter corresponding to alpha in the above formula.

         Beta: This required property is a floating-point parameter corresponding to beta in the above formula.

         Offset: This optional property is a floating-point parameter corresponding to k in the above formula. It defaults to 1.

As an example, consider the following response normalization bundle:

hidden RN1 [5, 10, 10]  
  from P1 response norm {  
    InputShape  = [ 5, 12, 12];  
    KernelShape = [ 1,  3,  3];  
    Alpha = 0.001;  
    Beta = 0.75;  
  }

The source layer includes five maps, each of dimension 12x12 totaling in 1440 nodes.

The value of KernelShape indicates that this is a same map normalization layer, where the neighborhood is a 3x3 rectangle.

The default value of Padding is false, thus the destination layer has only 10 nodes in each dimension. In order to include one node in the destination layer corresponding to every node in the source layer, add Padding=true and change the size of RN1 to [5, 12, 12].

### Share Declaration

Net# supports defining multiple bundles with shared weights. The weights of any two bundles can be shared as long as their structures are the same. Here is the syntax for defining bundles with shared weights:

share-declaration:  
share    {    layer-list    }  
share    {    bundle-list    }  
share    {    bias-list    }

layer-list:  
layer-name    ,    layer-name  
layer-list    ,    layer-name

bundle-list:  
bundle-spec    ,    bundle-spec  
bundle-list    ,    bundle-spec

bundle-spec:  
layer-name    =>     layer-name

bias-list:  
bias-spec    ,    bias-spec  
bias-list    ,    bias-spec

bias-spec:  
1    =>    layer-name

layer-name:  
identifier

For example:

const {  
  InputSize = 37;  
  HiddenSize = 50;  
}

input {  
  Data1 [InputSize];  
  Data2 [InputSize];  
}

hidden {  
  H1 [HiddenSize] from Data1 all;  
  H2 [HiddenSize] from Data2 all;  
}

output Result [2] {  
  from H1 all;  
  from H2 all;  
}

share { H1, H2 } // share both weights and biases

The input features are partitioned into two equal sized input layers. The hidden layers then compute higher level features on the two input layers. The share-declaration specifies that H1 and H2 must be computed in the same way from their respective inputs. This share-declaration simply specifies the layer names, indicating that both weights and biases should be shared. Alternatively, this could be specified with two separate share-declarations:

share { Data1 => H1, Data2 => H2 } // share weights

share { 1 => H1, 1 => H2 } // share biases

The shorthand form is only legal when the layers contain only a single bundle.

In general, sharing is only legal when the relevant structure is identical (same sizes, same convolutional geometry, etc).

## Expressions

Net# supports general expressions according to the grammar detailed in this section.

expression:  
or-expression

### Boolean operators

or-expression:  
and-expression  
or-expression    or    and-expression  
or-expression    ||    and-expression

and-expression:  
comparison-expression  
and-expression    and    comparison-expression  
and-expression    &&    comparison-expression

The operands of the Boolean operators must be of type Boolean. These operators use short-circuiting logic. The two forms of these operators (keyword versus punctuator) are semantically identical.

### Comparison operators

comparison-expression:  
after-comp  
equal-chain  
not-equal-chain  
increasing-chain  
decreasing-chain

The various forms of comparison-expression consist of alternating operands and operators. The forms are distinguished by the operators allowed.

equal-chain:  
after-comp    =    after-comp  
after-comp    ==    after-comp  
equal-chain    =    after-comp  
equal-chain    ==    after-comp

An equal-chain is two or more after-comps separated by the = or == operator. The two forms of the operator are semantically identical. If the operands evaluate to the same value, the result is true.

not-equal-chain:  
after-comp    <>    after-comp  
after-comp    !=    after-comp  
not-equal-chain    <>    after-comp  
not-equal-chain   !=    after-comp

A not-equal-chain is two or more after-comps separated by the <> or != operator. The two forms of the operator are semantically identical. If the operands evaluate to distinct values, the result is true. Note that since != is not transitive, a != b != c is not equivalent to a != b && b != c, but to a != b && b != c && a != c.

increasing-chain:  
after-comp    <    after-comp  
increasing-chain    <    after-comp  
after-comp    <=    after-comp  
increasing-chain    <=    after-comp

An increasing-chain is two or more after-comps separated by < or <= operators. If all pairs of adjacent operands satisfy the relationship indicated by the corresponding operator, the result is true.

decreasing-chain:  
after-comp    >    after-comp  
decreasing-chain    >    after-comp  
after-comp    >=    after-comp  
decreasing-chain    >=    after-comp

A decreasing-chain is analogous to an increasing-chain but with the directional comparisons reversed.

The increasing-chain and decreasing-chain operands must be numeric, either Integer or Float.

These operators provide expressivity closer to standard mathematical notation than is typical in standard programming languages. For example:

input Data [10];

output Out [10] from Data where (s,d) => 0 <= s - d < 3;

The predicate expression declares that there is a connection from source node s to destination node d iff the difference s - d is non-negative and less than 3. Without the chaining feature of these operators, this would need to be expressed as 0 <= s - d && s - d < 3 or d <= s && s < d + 3.

### Concatenation operator

after-comp:  
concatenation-expression

concatenation-expression:  
additive-expression  
concatenation-expression    &    additive-expression

Concatenation operates on tuples. Because tuples are automatically flattened, A & B is equivalent to [A, B].

### Addition

additive-expression:  
multiplicative-expression  
additive-expression     +    multiplicative-expression  
additive-expression     -    multiplicative-expression

### Multiplication

multiplicative-expression:  
prefix-unary-expression  
multiplicative-expression    \*    prefix-unary-expression  
multiplicative-expression    /    prefix-unary-expression  
multiplicative-expression    %    prefix-unary-expression

The semantics of division mimic those in C#. In particular, when both operands are Integers, the result is also an Integer and is the truncation of the mathematical result.

### Prefix unary operators

prefix-unary-expression:  
power-expression  
-    prefix-unary-expression  
!    prefix-unary-expression  
not    prefix-unary-expression

Both ! and not indicate Boolean negation.

### Exponentiation

power-expression:  
postfix-expression  
postfix-expression    ^    prefix-unary-expression

The exponentiation operator is right associative, accepting a prefix-unary-expression as its right operand.

### Postfix operators

postfix-expression:  
primary-expression  
postfix-expression    [    index    ]  
postfix-expression    [    index    :    count    ]

index:  
expression

count:  
expression

These express tuple indexing and sub-tuple extraction, respectively. For sub-tuple extraction, the first operand is the starting index, and the second operand is the length of the resulting tuple. Note that empty sub-tuples are illegal and that tuple extraction of size one is equivalent to indexing. Since count defines the result type of the sub-tuple extraction, and Net# is strongly typed, count must reduce to a constant value; in particular, count cannot depend on a parameter to a predicate.

### Primary expressions

primary-expression:  
numeric-literal  
string-literal  
boolean-literal  
identifier  
(    expression    )  
tuple  
invocation

Numeric-literal, string-literal, boolean-literal, and identifier are all non-terminals in the lexical grammar and are essentially the same as for the C# language.

An identifier may reference a declared constant or a parameter to a predicate.

tuple:  
[    expressions    ]

expressions:  
expression  
expressions    ,    expression

This tuple syntax builds a tuple from the given expressions. Tuples are always flattened, so if any of the expressions are themselves tuples, this concatenates them into the result. Similarly, if the result contains a single primitive value, the result is the primitive value (since tuples of length one are equivalent to the element).

### Invocation

invocation:  
function    (    expressionsopt    )

function:  
identifier

Net# supports only pre-defined functions. Currently there are three pre-defined functions, abs, param, and bittest.

The abs(x) function computes the absolute value of x. It is polymorphic, working on both Integer and Float operands.

The param(name, x) function takes a String name as its first operand and a default value as its second operand. The second operand is optional. If there is a parameter (provided by the host environment) with the given name, then this returns the corresponding value. Otherwise, if a default is specified, the default is returned. Otherwise, a compilation error is issued. The current parameters injected by the TLC environment are "InputNodeCount", "HiddenNodeCount", and "OutputNodeCount".

The bittest(bits, index) function takes a tuple of Integers as its first operand and an Integer bit index as its second operand. The index must be non-negative and less than 32 times the length of bits. Note that Net# supports hexadecimal notation, and it is typical to use hexadecimal for the items of bits.

The “save as text” function of the TLC Neural Network predictor dumps the output as Net#. For connection bundles that are neither full nor convolutional, the saved form is a filtered bundle with the predicate expression expressed as an invocation of bittest. For example, the net:

input Data [8];

output Out [8] from Data where (s,d) => abs(s - d) <= 1;

is saved as:

input Data [8];

output Out [8] from Data  
  where (s,d) => bittest(Out\_0\_Bits, s + d \* 8);

const Out\_0\_Bits = [ 0x1C0E0703, 0xC0E07038 ];

Note that the pruning feature of the Neural Network predictor can change full connection bundles into filtered bundles as training progresses.

# TLC API Tutorial

TLC offers a simple and intuitive interface to train models and predict using the API. There are 3 major interfaces, all included in Microsoft.MachineLearning.Core.dll:

* ITrainerFactory – this is the factory class for creating trainers and their args.
* ITrainer – This defines the actual training interface. Classes implementing this interface must implement the methods Train() and CreatePredictor().
* IPredictor – This defines the interface used for prediction. A predictor encapsulates the trained model and is used at run-time to make predictions. Classes implementing this interface must implement the method Predict().

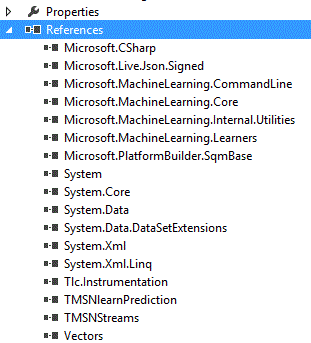
The interfaces mentioned above also have generic versions:

* ITrainerFactory<in TDataSet, out TPredictor> - where TDataSet is the class storing the training set used by the trainer, and TPredictor is the type of predictor the trainer creates.
* ITrainer<in TDataSet, out TPredictor> - where TDataSet is the class storing the training set used by the trainer, and TPredictor is the type of predictor the trainer creates.
* IPredictor<in TFeatures, out TResult> - where TFeatures is the class storing the features of an example, and TResult is the type the predictor returns as a prediction (in the existing predictors it is float for binary classification, regression and ranking and float[] for multi-class classification). IDistributionPredictor<in TFeatures, TResult, out TResultDistribution> is a specialization of IPredictor<in TFeatures, out TResult> and is used for predicting probabilities.

## Setup

The relevant assemblies are downloaded with the TLC package from <http://tlc>. These include:

* Microsoft.MachineLearning.Core.dll (contains the interfaces mentioned in the overview)
* TMSNlearnPrediction.dll (contains the various Instances classes, the various learners and some utility classes)
* Microsoft.MachineLearning.Learners.dll (contains the TLC learners)
* Microsoft.MachineLearning.CommandLine.dll (contains classes that enable command line parsing of arguments)
* Vectors.dll (contains classes defining mathematical structures that store instance features, and mathematical operations on these structures)
* Microsoft.Live.Json.Signed.dll, Microsoft.MachineLearning.Internal.Utilities.dll, Microsoft.PlatformBuilder.SqmBase.dll, Tlc.Instrumentation.dll and TMSNStreams.dll should also be referenced.For some learners, additional learner-specific assemblies are required.
* If native dlls are used (such as FastTreeNative.dll), they cannot be referenced, and they must be explicitly placed in the same directory as the program. Alternatively, they can be added using a post-build step, such as copy <path>/FastTreeNative.dll <output directory>.



## Using existing trainers and predictors

Create a project named TrainAndPredictUsingTLCLearners.

### Building a console application that trains and saves a model

First we will use the Linear SVM algorithm to train a model. In order to do so, add the needed [references](#tlc_api_tutorial_setup_htm) to your project.

We will use the file [\\cloudmltlc\TLC\Samples\breast-cancer.txt](file://cloudmltlc/tlc/samples/breast-cancer.txt) as a training set.

The trainer and its arguments are always created using a factory.

The training set is represented by a class implementing the interface Instances, in our case a TlcTextInstances object.

The model is trained by calling the method Train().

The model is saved using the static PredictorUtils class. This class has methods that save the model in the various formats: binary format, text format, INI format or code format. A model summary can also be saved. All types of models can be saved in binary format, which is the format used to load the model by TLC. Not all models can be saved in all other formats, please refer [here](#output_model_formats_htm) for details on which learners support which formats.

using Microsoft.MachineLearning;

using Microsoft.MachineLearning.CommandLine;

using Microsoft.MachineLearning.Learners;

using Microsoft.MachineLearning.Model;

using Microsoft.TMSN.TMSNlearn;

namespace TrainModel

{

    class Program

    {

        static void TrainAndSaveModel()

        {

            var factory = LinearSVMFactory.Instance;

            // create an arguments object with default values

            var arguments = factory.CreateArguments();

            // create the trainer

            var trainer = factory.CreateTrainer(arguments, new TrainHost(new Random(), 1));

            // load the training set

            // enter the correct path for the dataset:

            string dataSetName = @"\\cloudmltlc\TLC\Samples\breast-cancer.txt";

            // create the arguments object for the instances (for this dataset all the arguments are default)

            TlcTextInstances.Arguments instancesArgs = new TlcTextInstances.Arguments();

            // create the Instances object out of the dataset

            Instances instances = new TlcTextInstances(instancesArgs, dataSetName, null);

            // train the classifier

            trainer.Train(instances);

            // after the model is trained, the trainer can create the predictor

            var predictor = trainer.CreatePredictor();

            // save the model

            PredictorUtils.Save("svm-breast-cancer.model", predictor, instances.GetDataModel(), null, null);

            PredictorUtils.SaveSummary(predictor, "svm-breast-cancer.summary.txt");

            PredictorUtils.SaveText(predictor, instances.Schema.FeatureNames, "svm-breast-cancer.model.txt");

            PredictorUtils.SaveIni(predictor, instances.Schema.FeatureNames, "svm-breast-cancer.model.ini");

            PredictorUtils.SaveCode(predictor, instances.Schema.FeatureNames, "svm-breast-cancer.model.cs");

        }

        static void Main(string[] args)

        {

            TrainAndSaveModel();

        }

    }

}

### Loading and using trained models to predict

If a model has been trained and saved during a previous run, it can be loaded and used for prediction. Loading the model is done using the PredictorUtils class.

When a dataset contains categorical features or text features, the resulting numeric features of the instances depend on the training set (see the [categorical](#input_data_format_text_input_for_9140) and [text](#input_data_format_text_input_for_6513) feature sections for details). Therefore, when the Instances object is created from the test set, it may result in a different order of the features, a different number of features or even different features altogether, which would cause the predictor to fail. The numeric features and their order as created from the training set are saved together with the model, therefore when loading instances for testing the predictor must be passed as an argument. There are two ways to load test instances.

#### Loading test instances using RunExperiments.CreateTestData()

 The first is to create an Instances object using the static method RunExperiments.CreateTestData().

        static void Main(string[] args)

        {

            TrainAndSaveModel();

            string modelName = "svm-breast-cancer.model";

                          IDataModel dataModel;

                  IDataStats dataStats;

            var predictor = PredictorUtils.LoadPredictor<float>(out dataModel, out dataStats, modelName);

            if (predictor == null)

            {

                throw new Exception("Predictor is not a binary classifier");

            }

            // use the model to predict

            // enter the correct path for the dataset:

            string testSetName = @"\\cloudmltlc\TLC\Samples\breast-cancer.txt";

            // load the test set as an Instances object

            TLCArguments cmd = new TLCArguments();

            Instances testInstances = RunExperiments.CreateTestData(cmd, testSetName, dataModel);

            int total = 0;

            int correct = 0;

            // predict the class of each instance

            foreach (Instance instance in testInstances)

            {

                float prediction = predictor.Predict(instance) < 0 ? 0 : 1;

                if (instance.Label == prediction)

                    correct++;

                total++;

            }

            Console.WriteLine("{0} correct predictions out of {1}", correct, total);

        }

#### Loading test instances using TlcTextInstances

Loading test instances using TlcTextInstances is done by passing the information (saved in the predictor) about the features and their ordering. This is referred to as constructing the Instances object in *factory mode*, and it is indicated by passing null as the file name. The instances are then created one by one by calling the method ProduceInstance(). This method has two inputs. The first is a number representing the index of the instance, and is used only for the purpose of error notification. The second is the instance represented as a string. This way of loading the instances is useful in cases where the test instances are not in a file, for example, if they arrive online as input from the user.

Instead of these lines:

            // load the test set as an Instances object

            TLCArguments cmd = new TLCArguments();

            Instances testInstances = RunExperiments.CreateTestData(cmd, testSetName, dataModel);

Add the following code to the Main method:

            // load the test set as an Instances object

            var testInstancesArgs = new TlcTextInstances.Arguments();

            IInstanceFactory instanceFactory = new TlcTextInstances(testInstancesArgs, null, dataModel);

and replace the foreach loop with the new loop:

            // predict the class of each instance

            foreach (var instanceText in System.IO.File.ReadAllLines(testSetName))

            {

                Instance instance = instanceFactory.ProduceInstance(total, instanceText);

                float prediction = predictor.Predict(instance) < 0 ? 0 : 1;

                if (instance.Label == prediction)

                    correct++;

                total++;

            }

#### Loading Instances from in memory arrays

The Predict() method accepts an Instance as input, therefore if the instances are in memory as float arrays, predictions can be made as follows:

            Instance instance = new Instance (floatArrayInstance); // No Label in array, v-mingdi, no sparse representation.

            float prediction = predictor.Predict(instance) < 0 ? 0 : 1;

### Predicting probabilities

In addition to <in TFeatures, out TResult>, the TLC binary classifiers implement the interface IDistributionPredictor<in TFeatures, TResult, outTResultDistribution>. To use this option, after creating the predictor, add the following line after defining predictor:

            var stronglyTypedProbabilityPredictor = predictor as IDistributionPredictor<Instance, float, float>;

and replace the line

                float prediction = predictor.Predict(instance) < 0 ? 0 : 1;

with the lines

                float prediction;

                float probability;

                if (stronglyTypedProbabilityPredictor!=null)

                    probability = stronglyTypedProbabilityPredictor.PredictDistribution(instance, out prediction);

                else

                    prediction = predictor.Predict(instance);

### Predicting for multi-class

To make multi-class predictions, load the predictor as a multi-class predictor:

            var predictor = PredictorUtils.LoadPredictor<float[]>(modelName);

            if (predictor == null)

            {

                throw new Exception("Predictor is not a multi-class classifier");

            }

Then make predictions as follows:

                float[] prediction = predictor.Predict(instance);

### Choosing the learner based on user input

In case the learning algorithm is not known in advance, the factory can be created using the static class ReflectionUtils. Change the method TrainAndSaveModel() to take a string as an argument, and create the factory as follows:

        static void TrainAndSaveModel(string learnerName)

        {

            var factory = ReflectionUtils.FindFactory<float>(learnerName);

Change the call to TrainAndSaveModel in Main(string[] args) to

            TrainAndSaveModel(args[0]);

### Using non default arguments

Each TLC learner has some user defined parameters that can be passed as arguments to the algorithm. Also, some datasets are not in standard format, and therefore their format needs to be passed to the parser to create the Instances object. We use the static class CmdParser (located in Microsoft.MachineLearning.CommandLine.dll )to parse different kinds of arguments. To use this class add the line

using Microsoft.MachineLearning.CommandLine;

before the namespace definition.

We will use the file [\\cloudmltlc\TLC\Samples\UCI\adult.train](file://cloudmltlc/tlc/samples/uci/adult.train) as a training set and the file adult.test (in the samefolder) as a test set.

To train a model using the adult.train dataset, replace the code

            // enter the correct path for the dataset:

            string dataSetName = @"\\cloudmltlc\TLC\Samples\breast-cancer.txt";

 with the code

            // enter the correct path for the dataset:

            string dataSetName = @"\\cloudmltlc\TLC\Samples\UCI\adult.train";

After the line

            TlcTextInstances.Arguments instancesArgs = new TlcTextInstances.Arguments();

add the lines

            string instanceSettings = "header=+ label=14 cat=1,3,5-9,13 sep=,";

            CmdParser.ParseArguments(instanceSettings, instancesArgs);

To test a model on the adult.test dataset, replace the code

            // enter the correct path for the dataset:

            string testSetName = @"\\cloudmltlc\TLC\Samples\breast-cancer.txt";

with the code

            // enter the correct path for the dataset:

            string dataSetName = @"\\cloudmltlc\TLC\Samples\UCI\adult.test";

If using RunExperiments.CreateTestData(), after the line

            // load the test set as an Instances object

            TLCArguments cmd = new TLCArguments();

add the line

            cmd.instancesSettings = "header=+ label=14 cat=1,3,5-9,13 sep=,";

If using TlcTextInstances in instance factory mode, after the line

            var testInstancesArgs = new TlcTextInstances.Arguments();

add the lines

   stringinstanceSettings = "header=+ label=14 cat=1,3,5-9,13 sep=,";

           CmdParser.ParseArguments(instanceSettings, instancesArgs);

To use different values for the parameters of the Linear SVM algorithm, add these lines after the line creating the trainer arguments:

            string trainerArgs = "iter=20000 lr=0.005";

            CmdParser.ParseArguments(trainerArgs, arguments);

### Using the Native API to load TLC predictors

To deploy a neural network trained model, the model needs to be saved during training with the [/sbin](#tlc_learners_neural_networks_htm_5008) option. To consume the model for prediction:

* Download the stand-alone C++ library project called Predictor.Native from TLC source (or get it from TLC's TFS).
* Include the header Predictor.h file in your deployment project. Here's the predictor's interface:  
  template <typename TOut>  
  class Predictor  
  {  
  public:  
      int GetNumFeatures() { return NumFeatures; }  
      virtual TOut Predict(const float\* features) = 0;  
      virtual~Predictor() {}  
  protected:  
      int NumFeatures;  
  };  
  class Regressor : public Predictor<float>  
  {  
  public:  
      static Regressor\* Load(const string &modelFile);  
      static Regressor\* Load(istream& stream);  
  };  
  class BinaryClassifier : public Predictor<float>  
  {  
  public:  
      static BinaryClassifier\* Load(const string &modelFile);  
      static BinaryClassifier\* Load(istream& stream);  
  };  
  class MulticlassClassifier : public Predictor<Vec>  
  {  
  public:  
      static MulticlassClassifier\* Load(const string &modelFile);  
      static MulticlassClassifier\* Load(istream& stream);  
      int GetNumClasses() { return NumClasses; }  
  protected:  
      int NumClasses;  
  };
* Use Predictor::Load to construct the Predictor object from a model file or from a file stream.
* Call predictor->Predict(const float\* features) for prediction.

The predictor can also be used in a multi-threaded environment by loading the predictor from the model file and creating copies of the predictor (via the default copy constructor). Note that only the buffers are copied; the model parameters are not.

## Creating new trainers and predictors

As mentioned in the first section, building a learner involves implementing three interfaces: ITrainerFactory, ITrainer and IPredictor. As an example, let us create a simple k-nearest-neighbor learner (for more details about this algorithm, check out its [Wikipedia entry](http://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm)). Start by creating a new class library project named KNearestNeighbor and adding references to the [needed dll’s](#tlc_api_tutorial_setup_htm).

### Creating a predictor

Even though the training phase precedes the predicting phase, we start by implementing the predictor (since the trainer uses the predictor in its CreatePredictor() method). Add a new class named KNearestNeighborPredictor. Although the predictor can implement the interface IPredictor<in TFeatures, out TResult> directly, we let it derive from the abstract class PredictorBase<TOutput> (this class is in TMSNlearnPrediction.dll). TOutput is the type the Predict() method returns, and since we are building a binary classifier, TOutput is float.

using Microsoft.MachineLearning;

using Microsoft.TMSN.TMSNlearn;

namespace KNearestNeighbor

{

    public class KNearestNeighborPredictor : PredictorBase<float>

    {

        public override float Predict(Instance instance)

        {

            throw new NotImplementedException();

        }

        public override PredictionKind PredictionKind

        {

            get { throw new NotImplementedException(); }

        }

    }

}

The only parameter needed by the k-nearest-neighbor predictor is the value of k. It then compares the given instance to the whole training set, and picks the k nearest neighbors to it. Therefore, the predictor should keep a copy of the training set. We store the training set as two separate arrays: one for the feature vectors and one for the labels. The feature vectors are stored as an object of type Vector, located in the assembly Vectors.dll, in the namespace Microsoft.MachineLearning.Numeric. We also add an internal constructor to be called by the trainer in CreatePredictor().

using Microsoft.MachineLearning;

using Microsoft.TMSN.TMSNlearn;

using Microsoft.MachineLearning.Numeric;

namespace KNearestNeighbor

{

    public class KNearestNeighborPredictor : PredictorBase<float>

    {

        private Vector[] \_instances;

        private float[] \_labels;

        private int \_k;

        internal KNearestNeighborPredictor(int k, Instances instances)

        {

            \_k = k;

            \_instances = new Vector[instances.Count()];

            \_labels = new float[instances.Count()];

            int i = 0;

            foreach (Instance instance in instances)

            {

                \_instances[i] = instance.Features;

                \_labels[i++] = instance.Label;

            }

        }

        public override float Predict(Instance instance)

        {

            throw new NotImplementedException();

        }

        public override PredictionKind PredictionKind

        {

            get { throw new NotImplementedException(); }

        }

    }

}

Here is the full implementation that includes the Predict() method:

using Microsoft.MachineLearning;

using Microsoft.TMSN.TMSNlearn;

using Microsoft.MachineLearning.Numeric;

namespace KNearestNeighbor

{

    public class KNearestNeighborPredictor : PredictorBase<float>

    {

        private Vector[] \_instances;

        private float[] \_labels;

        private int \_k;

        internal KNearestNeighborPredictor(int k, Instances instances)

        {

            \_k = k;

            \_instances = new Vector[instances.Count()];

            \_labels = new float[instances.Count()];

            int i = 0;

            foreach (Instance instance in instances)

            {

                \_instances[i] = instance.Features;

                \_labels[i++] = instance.Label;

            }

        }

        public override float Predict(Instance instance)

        {

            // combine the array of feature vectors and their labels to one object (using "Select")

            // order the training instances by their distance from instance (using "OrderBy")

            // take the first k (using "Take")

            // count how many of the first k are labeled 1 (using "Count")

            int numberOf1s = \_instances.Select((featureVector, i) => new { Features = featureVector,

Label = \_labels[i] }).OrderBy(instance => Vector.Distance(features.Features, instance.Features)).Take(\_k).Count(instance => instance.Label > 0);

            return 2 \* numberOf1s > \_k ? 1 : 0;

        }

        public override PredictionKind PredictionKind

        {

            get { return PredictionKind.BinaryClassification; }

        }

    }

}

#### Implementing IDistributionPredictor

The k-nearest-neighbor algorithm can also predict the probability of belonging to class 1 by calculating the fraction of class 1 neighbors among the k nearest neighbors. Have the KNearestNeighborPredictor implement the interface IDistributionPredictor<Instance, float, float>:

using Microsoft.MachineLearning;

using Microsoft.TMSN.TMSNlearn;

using Microsoft.MachineLearning.Numeric;

namespace KNearestNeighbor

{

    public class KNearestNeighborPredictor : PredictorBase<float>, IDistributionPredictor<Instance, float, float>

    {

        private Vector[] \_instances;

        private float[] \_labels;

        private int \_k;

        internal KNearestNeighborPredictor(int k, Instances instances)

        {

            \_k = k;

            \_instances = new Vector[instances.Count()];

            \_labels = new float[instances.Count()];

            int i = 0;

            foreach (Instance instance in instances)

            {

                \_instances[i] = instance.Features;

                \_labels[i++] = instance.Label;

            }

        }

        public override float Predict(Instance instance)

        {

            // combine the array of feature vectors and their labels to one object (using "Select")

            // order the training instances by their distance from instance (using "OrderBy")

            // take the first k (using "Take")

            // count how many of the first k are labeled 1 (using "Count")

            int numberOf1s = \_instances.Select((featureVector, i) => new { Features = featureVector,

Label = \_labels[i] }).OrderBy(instance => Vector.Distance(features.Features, instance.Features)).Take(\_k).Count(instance => instance.Label > 0);

            return 2 \* numberOf1s > \_k ? 1 : 0;

        }

        public override PredictionKind PredictionKind

        {

            get { return PredictionKind.BinaryClassification; }

        }

        public float PredictDistribution(Instance features, out float result)

        {

            throw new NotImplementedException();

        }

        public float PredictDistribution(Instance features)

        {

            throw new NotImplementedException();

        }

    }

}

We move the computation of the number of class 1 neighbors to the method PredictDistribution() and let the method Predict() call PredictDistribution():

        public override float Predict(Instance instance)

        {

            float result;

            PredictDistribution(instance, outresult);

            return result;

        }

        public float PredictDistribution(Instance features, out float result)

        {

            // combine the array of feature vectors and their labels to one object (using "Select")

            // order the training instances by their distance from instance (using "OrderBy")

            // take the first k (using "Take")

            // count how many of the first k are labeled 1 (using "Count")

            int numberOf1s = \_instances.Select((featureVector, i) => new { Features = featureVector,

Label = \_labels[i] }).OrderBy(instance => Vector.Distance(features.Features, instance.Features)).Take(\_k).Count(instance => instance.Label > 0);

            result = 2 \* numberOf1s > \_k ? 1 : 0;

            return (float)numberOf1s / (float)\_k;

        }

        public float PredictDistribution(Instance features)

        {

            float result;

            return PredictDistribution(features, out result);

        }

#### Binary serialization of predictors

In order to save the k-nearest neighbor predictor in a format that is readable by TLC, it must implement the interface ICanSaveModel

### Creating a trainer

Next, we implement the trainer. Add a new class named KNearestNeighborTrainer. Intead of implementing the ITrainer interface directly, we let it derive from the abstract class TrainerBase (this class is in TMSNlearnPrediction.dll). TrainerBase is a generic class with two generic types; the first is float since k-nearest-neighbor is a binary classifier and the second is the type of predictor created by it, in our case KNearestNeighborPredictor.

using Microsoft.MachineLearning;

using Microsoft.TMSN.TMSNlearn;

namespace KNearestNeighbor

{

    public class KNearestNeighborTrainer : TrainerBase<float, KNearestNeighborPredictor>

    {

        public override KNearestNeighborPredictor CreatePredictor()

        {

            throw new NotImplementedException();

        }

        protected override void Initialize(Instances instances)

        {

            throw new NotImplementedException();

        }

        protected override void InnerTrain(Instances instances)

        {

            throw new NotImplementedException();

        }

        public override PredictionKind PredictionKind

        {

            get { throw new NotImplementedException(); }

        }

    }

}

The trainer must have fields to save the value of k and the training set in order to create the predictor. We also add a constructor that takes an integer k and a host, which is an object containing a Random object, two TextWriters (one for output and one for error) and a verbosity value.

        private Instances \_instances;

        private int \_k;

        public KNearestNeighborTrainer(int k, ITrainerHost host)

            : base(host)

        {

            \_k = k;

        }

The implementation of CreatePredictor() calls the method PopulateExtraData() which is defined in the class TrainerBase, and sends information about the [features used for training](#tlc_api_tutorial_using_existing__7646) to the predictor.

        public override KNearestNeighborPredictor CreatePredictor()

        {

            var predictor = new KNearestNeighborPredictor(\_k, \_instances);

            PopulateExtraData(predictor.ExtraData);

            return predictor;

        }

In the case of the k-nearest-neighbors algorithm, the training part is easy, since it does nothing but save the training set to use during prediction. The Initialize() method and the InnerTrain() method are called consecutively in the Train() method of TrainerBase, so saving the training set can be done in either method.

        protected override void Initialize(Instances instances)

        {

            \_instances = instances;

        }

        protected override void InnerTrain(Instances instances)

        {

        }

        public override PredictionKind PredictionKind

        {

            get { return PredictionKind.BinaryClassification; }

        }

### Creating a trainer factory

Finally, we implement the trainer factory. Add a new class named KNearestNeighborTrainerFactory, implementing the interface ITrainerFactory.

using System.Threading;

using Microsoft.MachineLearning;

using Microsoft.TMSN.TMSNlearn;

using KNearestNeighbor;

namespace KNearestNeighbor

{

    public class KNearestNeighborTrainerFactory : ITrainerFactory<Instances, KNearestNeighborPredictor>

    {

        public ITrainer<Instances, KNearestNeighborPredictor> CreateTrainer(ITrainerArguments args, ITrainerHost host)

        {

            throw new NotImplementedException();

        }

        public ITrainerArguments CreateArguments()

        {

            throw new NotImplementedException();

        }

        ITrainer ITrainerFactory.CreateTrainer(ITrainerArguments args, ITrainerHost host)

        {

            throw new NotImplementedException();

        }

        public PredictionKind GetPredictionKind(ITrainerArguments args)

        {

            throw new NotImplementedException();

        }

        public Type GetPredictorType(ITrainerArguments args)

        {

            throw new NotImplementedException();

        }

        public Type GetTrainerType(ITrainerArguments args)

        {

            throw new NotImplementedException();

        }

        public PredictionKindMask PredictionKindMask

        {

            get { throw new NotImplementedException(); }

        }

        public string LoadName

        {

            get { throw new NotImplementedException(); }

        }

        public string UserName

        {

            get { throw new NotImplementedException(); }

        }

    }

}

In order for TLC to recognize this learner using reflection, add the following attribute to the assembly, before the namespace definition:

using KNearestNeighbor;

[assembly: TrainerFactory(typeof(KNearestNeighborTrainerFactory),

   KNearestNeighborTrainerFactory.UserNameValue,

    KNearestNeighborTrainerFactory.LoadNameValue,

    "knn")]

The parameters of the TrainerFactoryAttribute are the type of the factory, a user name string which is the name displayed to the user (for example, in the drop down list of learners in the GUI) and an array of load name strings, which are the strings recognized by the ReflectionUtils class that enable creating the trainer.

To implement the factory class, first add a constructor. The constructor of the factory ensures that there is never more than one instance of a factory created.

        // This ensures that there is never more than one instance of a Factory created.

        private volatile static KNearestNeighborTrainerFactory \_instance;

        public static KNearestNeighborTrainerFactory Instance

        {

            get

            {

                if(\_instance == null)

                    Interlocked.CompareExchange(ref \_instance, new KNearestNeighborTrainerFactory(), null);

                return \_instance;

            }

        }

        private KNearestNeighborTrainerFactory()

        {

        }

Let us implement the rest of the methods starting at the end. First, implement the LoadName and UserName properties:

        // These properties are used for reflection.

        public const string LoadNameValue = "KNNLearner";

        internal const string UserNameValue = "K-Neareset-Neighbor Learner";

        public string LoadName

        {

            get { return LoadNameValue; }

        }

        public string UserName

        {

            get { return UserNameValue; }

        }

Next, implement the methods that return the type of prediction, trainer and predictor created by this factory:

        public PredictionKind GetPredictionKind(ITrainerArguments args)

        {

            return PredictionKind.BinaryClassification;

        }

        public Type GetPredictorType(ITrainerArguments args)

        {

            return typeof(KNearestNeighborPredictor);

        }

        public Type GetTrainerType(ITrainerArguments args)

        {

            return typeof(KNearestNeighborTrainer);

        }

        public PredictionKindMask PredictionKindMask

        {

            get { return PredictionKindMask.BinaryClassification; }

        }

We now get to the main purpose of the trainer factory, which is creating the trainer. The CreateTrainer() method has two input arguments. The second is the host, which is passed to the constructor of the trainer. The first is an object implementing the interface ITrainerArguments, defined in Microsoft.MachineLearning.Core.dll. Since each trainer has a different set of arguments, they must be wrapped up in an arguments object. Therefore, create a new class named KNearestNeighborTrainerArgs. Each argument in the arguments class must have two attributes: ArgumentAttribute and TGUIAttribute. The former lets the command line parser recognize the argument, and the latter defines how the argument will appear in the GUI. TGUIAttribute is defined in TMSNlearnPrediction.dll and ArgumentAttribute is defined in Microsoft.MachineLearning.CommandLine.dll.

using Microsoft.MachineLearning;

using Microsoft.MachineLearning.CommandLine;

using Microsoft.TMSN.TMSNlearn;

namespace KNearestNeighbor

{

    public class KNearestNeighborTrainerArgs : ITrainerArguments

    {

        [Argument(ArgumentType.AtMostOnce, HelpText = "number of neighbors", ShortName = "k")]

        [TGUI(Highlight = true, Description = "the number of neighbors used to find label", SuggestedSweeps = "1-9;inc:2")]

        public int k = 1;

    }

}

We can now implement the rest of the methods of the trainer factory:

      public ITrainer<Instances, KNearestNeighborPredictor> CreateTrainer(ITrainerArguments args, ITrainerHost host)

        {

            var argsCur = (KNearestNeighborTrainerArgs)args;

            return new KNearestNeighborTrainer(argsCur.k, host);

        }

        // This method returns a new ITrainerArguments of the correct type, with default values.

        public ITrainerArguments CreateArguments()

        {

            return new KNearestNeighborTrainerArgs();

        }

        ITrainer ITrainerFactory.CreateTrainer(ITrainerArguments args, ITrainerHost host)

        {

            return CreateTrainer(args, host); ;

        }

### Adding SubComponent arguments

Assume we would like to have the option to define different distance measures to determine the k nearest neighbors. To do so, we create an interface named IDistance and let the trainer and predictor contain a field implementing this interface.

using Microsoft.MachineLearning;

using Microsoft.TMSN.Numeric;

namespace KNearestNeighbor

{

    public interface IDistance

    {

        float ComputeDistance(Vector x, Vector y);

    }

}

The simplest implementation of this interface is the L2 distance.

    public class L2Distance : IDistance

    {

        public float ComputeDistance(Vector x, Vector y)

        {

            return Vector.Distance(x, y);

        }

    }

In order to be discoverable by reflection we define for each class implementing IDistance the attribute LoadableClassAttribute. Defining this attribute enables the static ReflectionUtils class to create instances of this type. The constructor of the attribute takes the type of the class, its constructor arguments (null in this case), a user name string which is the name displayed to the user (for example, in a drop down list of options in the GUI) and an array of load name strings, which are the strings recognized by the ReflectionUtils class. For L2Distance, add the following before the namespace definition:

using KNearestNeighbor;

[assembly: LoadableClass(typeof(L2Distance), null, "L2 Distance", "L2Distance")]

The next step is adding an IDistance field to the k-nearest-neighbor predictor. Update the fields and the constructor:

        private Vector[] \_instances;

        private float[] \_labels;

        private int \_k;

        private IDistance \_distance;

        internal KNearestNeighborPredictor(int k, Instances instances, IDistance distance)

        {

            \_k = k;

            \_distance = distance;

            \_instances = new Vector[instances.Count()];

            \_labels = new float[instances.Count()];

            int i = 0;

            foreach (Instance instance in instances)

            {

                \_instances[i] = instance.Features;

                \_labels[i++] = instance.Label;

            }

        }

and update the method PredictDistribution() accordingly:

        public float PredictDistribution(Instance features, out float result)

        {

            // combine the array of feature vectors and their labels to one object (using "Select")

            // order the training instances by their distance from instance (using "OrderBy")

            // take the first k (using "Take")

            // count how many of the first k are labeled 1 (using "Count")

            int numberOf1s = \_instances.Select((featureVector, i) => new { Features = featureVector,

Label = \_labels[i] }).OrderBy(instance => \_distance.ComputeDistance(features.Features, instance.Features)).Take(\_k).Count(instance => instance.Label > 0);

            result = 2 \* numberOf1s > \_k ? 1 : 0;

            return (float)numberOf1s / (float)\_k;

        }

Finally, we update the trainer, the trainer arguments and the trainer factory. The ReflectionUtils class uses the method CreateInstance() to create objects of different types. The input parameter for this method is an object of type SubComponent (also defined in Microsoft.MachineLearning.CommandLine.dll), which encapsulates the type’s name and its argument values (if it has any). We therefore define the distance object as a SubComponent:

using Microsoft.MachineLearning;

using Microsoft.MachineLearning.CommandLine;

using Microsoft.TMSN.TMSNlearn;

namespace KNearestNeighbor

{

    public class KNearestNeighborTrainerArgs : ITrainerArguments

    {

        [Argument(ArgumentType.AtMostOnce, HelpText = "number of neighbors", ShortName = "k")]

        [TGUI(Highlight = true, Description = "the number of neighbors used to find label", SuggestedSweeps = "1-9;inc:2")]

        public int k = 1;

        [Argument(ArgumentType.Multiple, HelpText = "How to compute the distance between instances?", ShortName = "distance")]

        [TGUI(Highlight = true, Description = "how to compute the distance between instances?", BaseType = typeof(IDistance))]

        public SubComponent distanceClassName = new SubComponent("L2Distance");

    }

}

Update the trainer to have an instance of KNearestNeighborTrainerArgs as a field, and change the constructor accordingly:

        private Instances \_instances;

        private KNearestNeighborTrainerArgs \_args;

        public KNearestNeighborTrainer(KNearestNeighborTrainerArgs args, ITrainerHost host)

            : base(host)

        {

            \_args = args;

        }

Update the CreatePredictor() method of the trainer:

        public override KNearestNeighborPredictor CreatePredictor()

        {

            var predictor = new KNearestNeighborPredictor(\_args.k, \_instances, ReflectionUtils.CreateInstance<IDistance>(\_args.distanceClassName));

            PopulateExtraData(predictor.ExtraData);

            return predictor;

        }

The last change to be made is in the CreateTrainer() method of the trainer factory:

        public ITrainer<Instances, KNearestNeighborPredictor> CreateTrainer(ITrainerArguments args, ITrainerHost host)

        {

            var argsCur = (KNearestNeighborTrainerArgs)args;

            return new KNearestNeighborTrainer(argsCur, host);

        }

### Using SubComponents that have their own arguments

In order to create an instance of a class, ReflectionUtils must call the constructor. If the constructor of the class has input arguments, their type must be passed using the LoadableClassAttribute. This attribute has two constructors, one takes the type of an arguments class and the other takes the type of an arguments class and an array of the other constructor parameter types. In both, the type of arguments class may be null.

Let us define another class implementing the IDistance interface, named LpDistance.

    public class LpDistance : IDistance

    {

        private int \_p;

        public LpDistance(int p)

        {

            \_p = p;

        }

        public float ComputeDistance(Vector x, Vector y)

        {

            WritableVector diff = x - y;

            if (\_p == 1)

                return diff.L1Norm();

            float res=0;

            for (int i = 0; i < diff.Length; i++)

                res += (float)Math.Pow(diff[i], \_p);

            return (float)Math.Pow(res, 1 / \_p);

        }

    }

The LoadableClassAttribute for this class is defined as follows:

[assembly: LoadableClass(typeof(LpDistance), null, new Type[] { typeof(int) }, "Lp Distance", "LpDistance")]

Alternatively, an Arguments class can be used:

    public class LpDistance : IDistance

    {

        public sealed class Arguments

        {

            [Argument(ArgumentType.AtMostOnce, HelpText = "which norm to use?", ShortName = "p")]

            public int p = 1;

        }

        private int \_p;

        public LpDistance(Arguments args)

        {

            \_p = args.p;

        }

        public float ComputeDistance(Vector x, Vector y)

        {

            WritableVector diff = x - y;

            if (\_p == 1)

                return diff.L1Norm();

            float res=0;

            for (int i = 0; i < diff.Length; i++)

                res += (float)Math.Pow(diff[i], \_p);

            return (float)Math.Pow(res, 1 / \_p);

        }

    }

in which case the LoadableClassAttribute is defined as follows:

[assembly: LoadableClass(typeof(LpDistance), typeof(LpDistance.Arguments) , "Lp Distance", "LpDistance")]

To define L2 as the default distance measure using the class LpDistance, change the definition of distanceClassName in KNearestNeighborTrainerArgs as follows:

        [Argument(ArgumentType.Multiple, HelpText = "How to compute the distance between instances?", ShortName = "distance")]

        [TGUI(Highlight = true, Description = "how to compute the distance between instances?", BaseType = typeof(IDistance))]

        public SubComponent distanceClassName = new SubComponent("LpDistance", "p=2");

### Binary serialization of predictors

In order to save the k-nearest neighbor predictor in a format that is readable by TLC, it must implement either the ICanSaveInBinaryFormat interface, or the ICanSaveModel Interface. The latter is needed only in case the predictor contains objects such as normalizers or calibrators that the user may want to load and use separately. In this case, the types that are saved separately must implement ICanSaveInBinaryFormat.

Since KNearestNeighborPredictor contains a field of type IDistance, we let it implement ICanSaveModel. This interface is defined in the assembly TMSNlearnPrediction.dll in the namespace Microsoft.MachineLearning.Model. It has only one method:

        void Save(RepositoryWriter rep, string dir, string name);

This method is virtually implemented in PredictorBase<TOutput>:

        public virtual void Save(RepositoryWriter rep, string dir, string name)

        {

            ModelSaveContext.Save(rep, dir, name, SaveCore);

        }

The ModelSaveContext.Save method creates a zip folder containing the information needed to reload the predictor. SaveCore is also a virtual method implemented in PredictorBase<TOutput>, and it should be overridden by predictors deriving from it. The type ModelSaveContext contains a BinaryWriter that is used to write the information in the predictor. The Microsoft.MachineLearning.Internal.Utilities.dll assembly contains some extension methods to BinaryWriter, so add a reference to it, and add the following lines in the code:

usingMicrosoft.MachineLearning.Model;

usingMicrosoft.MachineLearning.Internal.Utilities;

Next, we update KNearestNeighborPredictor to implement ICanSaveModel:

    public class KNearestNeighborPredictor : PredictorBase<float>,

        IDistributionPredictor<Instance, float, float>,

        ICanSaveModel

    {

For backward compatibility, TLC binary serialization is versioned, therefore each class that can be serialized needs to have the following method, to be called during serialization:

        public const string LoaderSignature = "KNNExec";

        private static VersionInfo GetVersionInfo()

        {

            return new VersionInfo(

                modelSignature: "KNN PRED",

                verWrittenCur: 0x00010001, // Initial

                verReadableCur: 0x00010001,

                verWeCanReadBack: 0x00010001,

                loaderSignature: LoaderSignature);

        }

Here is the implementation of SaveCore for KNearestNeighborPredictor:

        protected override void SaveCore(ModelSaveContext ctx)

        {

            base.SaveCore(ctx);

            ctx.SetVersionInfo(GetVersionInfo());

            // \*\*\* Binary format \*\*\*

            // int: n (number of instances)

            // int: number of features

            // for i=0..n-1

            //   int[]: the i'th instance

            // float[]: the labels

            // int: \_k

            ctx.Writer.Write(\_instances.Length);

            ctx.Writer.Write(\_instances[0].Count);

            float[] values = new float[\_instances[0].Count];

            for (int i = 0; i < \_instances.Length; i++)

            {

                \_instances[i].GetValues(values);

                ctx.Writer.Write(values);

            }

            ctx.Writer.Write(\_labels);

            ctx.Writer.Write(\_k);

            // Save other streams.

            ctx.SaveModel(\_distance, @"Distance");

        }

The last line above, creates a directory called Distance in the zip file, which contains the binary serialization of the \_distance field. It is recommended that the type of the object specified as the first input argument of ctx.SaveModel implement ICanSaveInBinaryFormat or ICanSaveModel, since otherwise the non-versionable .NET serialization is used.

In order to deserialize the model, we implement the following two methods:

 private KNearestNeighborPredictor(ModelLoadContext ctx)

            : base(ctx)

        {

            // \*\*\* Binary format \*\*\*

            // int: n (number of instances)

            // int: number of features

            // for i=0..n-1

            //   int[]: the i'th instance

            // float[]: the labels

            // int: \_k

            int n = ctx.Reader.ReadInt32();

            Contracts.CheckDecode(n > 0);

            int numFeatures = ctx.Reader.ReadInt32();

            Contracts.CheckDecode(numFeatures > 0);

            \_instances = new Vector[n];

            for (int i = 0; i < n; i++)

            {

                var inst = ctx.Reader.ReadFloatArray();

                Contracts.CheckDecode(inst.Length == numFeatures);

                Contracts.CheckDecode(inst.All(x => FloatUtils.IsFinite(x)));

                \_instances[i] = new Vector(inst);

            }

            \_labels = ctx.Reader.ReadFloatArray();

            Contracts.CheckDecode(\_labels.Length == n);

            Contracts.CheckDecode(\_labels.All(x => FloatUtils.IsFinite(x)));

            \_k = ctx.Reader.ReadInt32();

            Contracts.CheckDecode(\_k > 0);

            // read distance

            ctx.LoadModel(out \_distance, @"Distance");

        }

        public static KNearestNeighborPredictor Create(RepositoryReader rep, Repository.Entry ent, string dir)

        {

            return ModelLoadContext.Load(rep, ent, dir, GetVersionInfo(), ctx => new KNearestNeighborPredictor(ctx));

        }

In order for TLC to load the k-nearest-neighbor model, the following attribute should be added:

[assembly: LoadableClass(typeof(KNearestNeighborPredictor), null, typeof(SignatureLoadModel),

    "K-Nearest-Nbr Executor",

    KNearestNeighborPredictor.LoaderSignature)]

#### Serialization of the IDistance field

As mentioned earlier, in order for the \_distance field to be serialized with versioning, we let the classes implementing IDistance implement the interface ICanSaveInBinaryFormat. To implement this interface add the following lines to the file containing the distance classes definitions:

usingMicrosoft.MachineLearning.Model;

using Microsoft.TMSN.TMSNlearn;

The L2Distance class has no fields it needs to save. As an example, here is how the ICanSaveInBinaryFormat interface is implemented:

    public class L2Distance : IDistance, ICanSaveInBinaryFormat

    {

        public const string LoaderSignature = "L2DistExec";

        private static VersionInfo GetVersionInfo()

        {

            return new VersionInfo(

                modelSignature: "L2 DSTNC",

                verWrittenCur: 0x00010001, // Initial

                verReadableCur: 0x00010001,

                verWeCanReadBack: 0x00010001,

                loaderSignature: LoaderSignature);

        }

        public L2Distance()

        {

        }

        public static L2Distance Create(System.IO.BinaryReader reader)

        {

            return ModelLoadContext.Load(reader, GetVersionInfo(), ctx => new L2Distance(ctx));

        }

        private L2Distance(ModelLoadContext ctx)

        {

            // \*\*\* Binary format \*\*\*

        }

        private void SaveCore(ModelSaveContext ctx)

        {

            Contracts.AssertValue(ctx);

            ctx.SetVersionInfo(GetVersionInfo());

            // \*\*\* Binary format \*\*\*

        }

        public void SaveAsBinary(System.IO.BinaryWriter writer)

        {

            ModelSaveContext.Save(writer, SaveCore);

        }

        public float ComputeDistance(Vector x, Vector y)

        {

            return Vector.Distance(x, y);

        }

    }

Notice that since we are adding a non-default constructor, the default constructor must be explicitly defined.

Here is the implementation of LpDistance:

    public class LpDistance : IDistance, ICanSaveInBinaryFormat

    {

        public sealed class Arguments

        {

            [Argument(ArgumentType.AtMostOnce, HelpText = "which norm to use?", ShortName = "p")]

            public int p = 1;

        }

        public const string LoaderSignature = "LpDistExec";

        private static VersionInfo GetVersionInfo()

        {

            return new VersionInfo(

                modelSignature: "LP DSTNC",

                verWrittenCur: 0x00010001, // Initial

                verReadableCur: 0x00010001,

                verWeCanReadBack: 0x00010001,

                loaderSignature: LoaderSignature);

        }

        private int \_p;

        public static LpDistance Create(System.IO.BinaryReader reader)

        {

            return ModelLoadContext.Load(reader, GetVersionInfo(), ctx => new LpDistance(ctx));

        }

        private LpDistance(ModelLoadContext ctx)

        {

            // \*\*\* Binary format \*\*\*

            // int: p

            \_p = ctx.Reader.ReadInt32();

            Contracts.CheckDecode(\_p > 0);

        }

        private void SaveCore(ModelSaveContext ctx)

        {

            Contracts.AssertValue(ctx);

            ctx.SetVersionInfo(GetVersionInfo());

            // \*\*\* Binary format \*\*\*

            // int: p

            ctx.Writer.Write(\_p);

        }

        public void SaveAsBinary(System.IO.BinaryWriter writer)

        {

            ModelSaveContext.Save(writer, SaveCore);

        }

        public LpDistance(Arguments args)

        {

            \_p = args.p;

        }

        public float ComputeDistance(Vector x, Vector y)

        {

            WritableVector diff = x - y;

            if (\_p == 1)

                return diff.L1Norm();

            float res=0;

            for (int i = 0; i < diff.Length; i++)

                res += (float)Math.Pow(diff[i], \_p);

            return (float)Math.Pow(res, 1 / \_p);

        }

    }

In order for TLC to load the distance objects, the following attribute should be added:

[assembly: LoadableClass(typeof(L2Distance), null, typeof(SignatureLoadBinary),

 "L2 distance Executor", L2Distance.LoaderSignature)]

[assembly: LoadableClass(typeof(LpDistance), null, typeof(SignatureLoadBinary),

 "Lp distance Executor", LpDistance.LoaderSignature)]

### Testing the new learner

To test the new learner, add a new console application project to the solution named TryKNearestNeighbor and set it as StartUp project. Add the references described in section ‎2, and also a reference to KnearestNeighbor. Copy the Main() method and the TrainAndSaveModel() method from the project TrainAndPredictUsingTLCLearners. Change “svm” in all the file names to “knn”.

To use the k-nearest-neighbor learner with the settings k=5 and distance measure L3, change the code as follows:

using Microsoft.MachineLearning;

using Microsoft.TMSN.TMSNlearn;

using Microsoft.MachineLearning.CommandLine;

using KNearestNeighbor;

namespace TryKNearestNeighbor

{

    class Program

    {

        static void TrainAndSaveModel()

        {

            var factory = KNearestNeighborTrainerFactory .Instance;

            // create an arguments object with default values

            var arguments = factory.CreateArguments();

            // create the trainer

            var trainer = factory.CreateTrainer(arguments, new TrainHost(new Random(), 1));

            // load the training set

            // enter the correct path for the dataset:

            string dataSetName = @"\\cloudmltlc\TLC\Samples\UCI\adult.train";

            // create the arguments object for the instances (for this dataset all the arguments are default)

            TlcTextInstances.Arguments instancesArgs = new TlcTextInstances.Arguments();

            string instanceSettings = "header=+ label=14 cat=1,3,5-9,13 sep=,";

            CmdParser.ParseArguments(instanceSettings, instancesArgs);

            // create the Instances object out of the dataset

            Instances instances = new TlcTextInstances(instancesArgs, dataSetName, null);

            // train the classifier

            trainer.Train(instances);

            // after the model is trained, the trainer can create the predictor

            var predictor = trainer.CreatePredictor();

            // save the model

            PredictorUtils.Save("knn-adult.model", predictor, instances.GetDataModel(), null, null);

            PredictorUtils.SaveSummary(predictor, "knn-adult.summary.txt");

            PredictorUtils.SaveText(predictor, instances.Schema.FeatureNames, "knn-adult.model.txt");

            PredictorUtils.SaveIni(predictor, instances.Schema.FeatureNames, "knn-adult.model.ini");

            PredictorUtils.SaveCode(predictor, instances.Schema.FeatureNames, "knn-adult.model.cs");

        }

        static void Main(string[] args)

        {

            TrainAndSaveModel();

            string modelName = "knn-adult.model";

                          IDataModel dataModel;

                  IDataStats dataStats;

            var predictor = PredictorUtils.LoadPredictor<float>(out dataModel, out dataStats, modelName);

            if (predictor == null)

            {

                throw new Exception("Predictor is not a binary classifier");

            }

            // use the model to predict

            // enter the correct path for the dataset:

            string testSetName = @"\\cloudmltlc\TLC\Samples\UCI\adult.test";

            TLCArguments cmd = new TLCArguments();

            cmd.instancesSettings = "header=+ label=14 cat=1,3,5-9,13 sep=,";

            Instances testInstances = RunExperiments.CreateTestData(cmd, testSetName, dataModel);

            int total = 0;

            int correct = 0;

            // predict the class of each instance

            foreach (Instance instance in testInstances)

            {

                if (instance.Label == predictor.Predict(instance))

                    correct++;

                total++;

                if (total % 100 == 0)

                    Console.WriteLine("correct:{0}, total:{1}", correct, total);

            }

            Console.WriteLine("{0} correct predictions out of {1}", correct, total);

        }

    }

}

# Index

A

Accuracy 7

Aether 5, 215

All-vs-all 6, 163

API 5

Attribute features 18

AUC 7

Averaged Perceptron 5, 89, 124

Azure 5, 61, 67, 73, 77, 203

B

Bagging 144

Binary classification 5, 107, 113, 124, 131, 136, 137, 144, 161

Binary input format 13, 16, 83

Binary model format 53

BinaryInstanceWriter 13, 83

Bing Extraction input format 43

Bing/OSD input format 43

Binning normalizer 91

Boosted trees 124, 137

Bootstrap aggregating 144

Bramble 5, 136

C

Calibration 5, 124, 131

Categorical features 18

Classification 5, 107, 124

Code model format 56

Command line examples 205

command-line 5, 13

Cosmos 5

Cross Validation 5, 61

Cross Validation folds 61

D

Data Formats 15, 83

DCG 11, 124

Decision Trees 89, 124, 137, 217

Discriminative GMM 136

E

Error Explorer 221

Evaluator 13, 61, 67

Examples 18, 49, 61, 67, 73, 77, 89, 205, 207

Experiment Results 209

Extra Assemblies 203

F

FastRank 5, 43, 85, 124, 217

FastTree 5, 43, 137, 217

Feature exclusion 18

Feature inclusion 18

Feature Normalization 83, 91, 109, 113, 124, 131, 168, 172

Feature Selection 5, 85

Feature Transformation 48

G

Gaussian normalizer 91

Grouping 10, 40

GUI 5, 13

GUI examples 207

H

HPC 5, 203

I

Image input format 45

ini model format 53

Instances 43, 45, 61, 67, 73, 77

Isotonic regression (PAV calibration) 5

L

L1 loss 9

L2 loss 9

Labels 18, 43

Learning mode 61

Learning rate 113, 124, 131, 172

Linear learners 89, 109, 124, 131, 168, 172

Logistic Regression 5, 85, 109

Log-loss 7

Loss function 113, 124, 172

M

Macro-average 7

Metrics 7, 9, 11

Micro-average 7

Min-Max normalizer 13, 91

Missing features 18

MLTK.HNN 136

Model summary 58, 85

Multi-class classification 5, 6, 107, 113, 136, 144, 163, 165

N

Naive Calibration 5

Name features 18

NDCG 11, 124

Net# 113, 223

Network Definition Language 113, 223

Neural Networks 5, 13, 89, 113, 165

O

One-vs-all 6, 163

Output Options 199

Overfitting 98

P

Pairwise coupling 6, 163

Parallel Ensemble 144

Parameter Sweeps 5, 99, 209

Patching 165

PAV calibration 5

PKPD 6, 163

Poisson regression 168

Precision and Recall 7, 199

R

Random forests 136

Ranker metrics 11

Ranking 5, 10, 40, 43, 107, 124, 137

Regression 5, 8, 107, 113, 124, 137, 161, 168, 172

Regularization 98, 165

Results Processor 5, 209

Root mean square loss 9

Run Options 203

S

Saving models 5, 67, 73

Sigmoid calibration 5

Sparse input format 18

Stochastic gradient descent 5, 172

Stratification 61, 161

Strato Learner 161

SVM 5, 89, 131

T

Test set 67, 77

Testing models 5, 61

Text features 18

Text input format 18, 83

Text model format 56

TextInstanceWriter 83

Threads 18, 109, 163, 168, 203

TLC model format 53, 77

Training models 5, 61

Training set 61, 67, 73

Tree Visualizer 5, 217

V

Vowpal Wabbit 5, 137

W

Weighted examples 18, 109