Submitted 5/17; published M/YY

Standard Machine Learning Language: A Language Agnostic Framework for Streamlining the Development of Machine Learning Pipelines

Kelechi Ikegwu IKEGWU2@ILLINOIS.EDU
Illinois Informatics Institute
The University of Illinois at Urbana-Champaign

Micheal Hao MXHAO2@ILLINOIS.EDU
Department of Electrical and Computer Engineering
The University of Illinois at Urbana-Champaign

Neeraj Asthana NEEASTHANA@GMAIL.COM
Department of Computer Science
Department of Statistics
The University of Illinois at Urbana-Champaign

Robert Brunner BIGDOG@ILLINOIS.EDU
Department of Accountancy
Department of Computer Science
Illinois Informatics Institute
School of Information Sciences
Department of Statistics
The University of Illinois at Urbana-Champaign

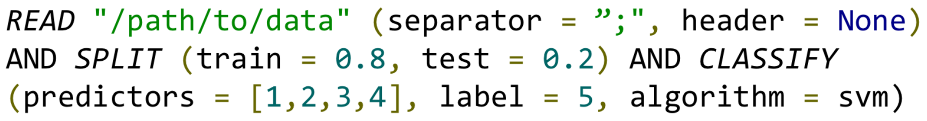
Abstract

Standard Machine Learning Language (SML) is a language agnostic framework that integrates a query-like language to simplify the development of machine learning pipelines. Emphasis was placed on ease of use and abstracting the complexities of machine learning from the end user encouraging its use in professional and academic settings for a variety of disciplines. SML's architecture is discussed, followed by multiple interfaces that one could use to interact with SML. Lastly, SML is applied to a few problems, and the complexities of SML query's and traditional approaches used to solve problems are compared and discussed.

1. Introduction

Machine Learning has simplified the process of solving a vast amount problems in a variety of fields by learning from data. In most cases, machine learning has become more attractive than manually creating programs to address these same issues. However there's a multitude of nuisances involved when developing machine learning pipelines (Domingos, 2012). If these nuisances are not taken into consideration one may not receive satisfactory results. A domain expert utilizing machine learning to solve problems may not want or have the time to deal with these complexities. To combat these issues we introduce Standard Machine Learning Language (SML).

The overall objective of the SML is to provide a level of abstraction which simplifies the development process of machine learning pipelines. Consequently this enables students,



```
READ "/path/to/data" (separator = ”;", header = None)
AND SPLIT (train = 0.8, test = 0.2) AND CLASSIFY
(predictors = [1,2,3,4], label = 5, algorithm = svm)
```

Figure 1: Example of a SML Query performing classification.

researchers, and industry professionals without a background in machine learning to solve problems in different domains with machine learning. We developed SML a query like language which serves as an abstraction from writing a lot of code (see Figure 1 for an example). In the subsequent sections related works are discussed followed by defining the grammar used to create queries for SML. The architecture of SML is described, lastly SML is applied to use-cases to demonstrate how it reduces the complexity of solving problems that utilize machine learning.

2. Related Works

They're related works that attempt to provide a level of abstraction as well for writing machine learning code. TPOT (Olson, Bartley, Urbanowicz, & Moore, 2016) is a tool implemented in Python that creates and optimizes machine learning pipelines using genetic programming. Given cleaned data, TPOT performs feature selection, preprocessing , and construction. Given the task (classification, regression, or clustering) it uses the best features to determine the most optimal model to use. Lastly, it performs optimization on parameters for the selected model. What differentiates SML from TPOT is that in addition to feature, model, and parameter selection/optimization a framework is in place to apply these models to different datasets and construct visualizations for different metrics with each algorithm.

LBJava (Rizzolo & Roth, 2010) is another tool based on a programming paradigm called Learning Based Programming (Roth, 2005) which is an extension of conventional programming that creates functions using data driven approaches. LBJava follows the principles of Learning Based Programming by abstracting the details of common machine learning processes. What separates SML from LBJava and TPOT is that it offers a higher level of abstraction by providing a query like language which allows people who aren't experienced programmers to use SML.

3. Grammar

The SML language is a domain specific language with grammar implemented in Bakus-Naur form (BNF). Each expression has a rule and can be expanded into other terms. Figure 1 is an example of how one would perform classification on a dataset using SML. The query in Figure 1 reads from a dataset, performs a 80/20 split of training and testing data respectively, and performs classification on the 5th column of the hypothetical dataset using columns 1,2,3, and 4 as predictors. In the subsequent subsections SML's grammar in BNF form is defined in addition to the keywords.

3.1 Grammar Structure

This subsection is dedicated to defining the grammar of SML in terms of BNF. A *Query* can be defined by a delimited list of actions where the delimiter is an AND statement; with BNF syntax this is defined as:

$$< Query >::=< Action > \mid < Action > \ AND < Query > \tag{1}$$

An *Action* in (1) follows one of the following structures defined in (2) where a $Keyword$ is required followed by an *Argument* and/or *OptionList*.

$$\begin{aligned} < Action >::=< Keyword > \ < Argument > \\ \mid < Keyword > \ < Argument > (< OptionList >) \\ \mid < Keyword > (< OptionList >) \end{aligned} \tag{2}$$

A *Keyword* is a predefined term associating an *Action* with a particular string. An *Argument* generally is a single string surrounded by quotes that specifies a path to a file. Lastly, an *Argument* can have a multitude of options (3) where an *Option* consist of an $OptionName$ with either an $OptionValue$ or $OptionValueList$. An $OptionName$, and $OptionValue$ consist of a single string, an *OptionList* (4) consist of a comma delimited list of options and an $OptionValueList$ (5) consist of a comma delimited list of $OptionValues$.

$$\begin{aligned} < Option >::=< OptionName > \ = \ < OptionValue > \\ \mid < OptionName > \ = [< OptionValueList >] \end{aligned} \tag{3}$$

$$< OptionList >::=< Option > \mid < Option > , \ < OptionList > \tag{4}$$

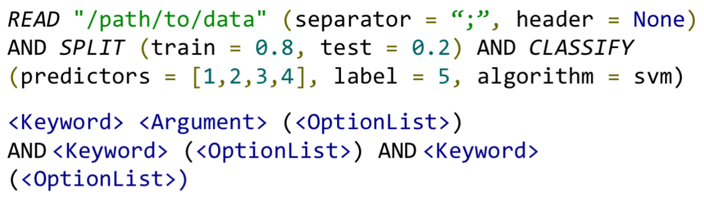
$$\begin{aligned} < OptionValueList >::=< OptionValue > \\ \mid < OptionValue > , \ < OptionValueList > \end{aligned} \tag{5}$$

To put the grammar into perspective the example *Query* in Figure 1 has been transcribed into BNF format and can be found in Figure 2. The next subsection describes the functionality for all *Keyword*s of SML.

3.2 Keywords

Currently there are 8 *Keyword*s in SML [1]. These *Keyword*s can be chained together to perform a variety of actions. In the subsequent subsections we describe the functionality of each *Keyword*.

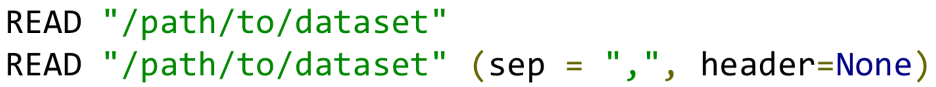
[1] Detailed documentation providing examples and describing all of the keywords of SML are publicly available on github: https://github.com/lcdm-uiuc/sml/tree/master/dataflows



```
READ "/path/to/data" (separator = “;”, header = None)
AND SPLIT (train = 0.8, test = 0.2) AND CLASSIFY
(predictors = [1,2,3,4], label = 5, algorithm = svm)

<Keyword> <Argument> (<OptionList>)
AND <Keyword> (<OptionList>) AND <Keyword>
(<OptionList>)
```

Figure 2: Here the example *Query* on the top was defined in Figure 1 and the bottom *Query* is in BNF format. For the example *Query* the first *Keyword* is *READ* followed by an *Arugment* that specifies the path to the dataset, next an *OptionValueList* containing information about the delimiter of the dataset and the header. We then include the *AND* delimiter to specify an additional *Keyword SPLIT* with an *OptionValueList* that tells us the size of the training and testing partitions for the dataset specified with the *READ Keyword*. Lastly, the *AND* delimiter is used to specify another *Keyword CLASSIFY* which performs classification using the training and testing data from the result of the *SPLIT Keyword* followed by an *OptionValueList* which provides information to SML about the features to use (columns 1-4), the label we want to predict (column 5), and the algorithm to use for classification.



```
READ "/path/to/dataset"
READ "/path/to/dataset" (sep = ",", header=None)
```

Figure 3: Example using the *READ Keyword* in SML.

3.2.1 Reading Datasets

When reading data from SML one must use the *READ Keyword* followed by an *Argument* containing a path to the dataset. *READ* also accepts a variety of *Options*. The first *Query* in Figure 3 consist of only a *Keyword* and *Argument*. This *Query* reads in data from ”/path/to/dataset”. The second *Query* includes an *OptionValueList* in addition to reading data from the specified path; the *OptionValueList* specifies that the dataset is delimited with semicolons and does not include a header row.

3.2.2 Cleaning Data

When NaNs, NAs and/or other troublesome values are present in dataset we clean these values in SML by using the *REPLACE Keyword*. Figure 4 shows an example of the *REPLACE Keyword* being used. In this *Query* we use the *REPLACE Keyword* in conjugation with the *READ Keyword*. SML reads from a comma delimited dataset with no header from the path ”/path/to/dataset”. Then we replace any instance of ”NaN” with the mode of that column in the dataset.

```
READ "/path/to/data" (separator = ",", header = None)
AND REPLACE (missing = "NaN",  strategy = "mode")
```

Figure 4: An example utilizing the *REPLACE Keyword* in SML.

```
READ "/path/to/data" (separator = ",", header = None) AND
SPLIT (train = 0.8, test = 0.2)
```

Figure 5: Example using the *SPLIT Keyword* in SML.

3.2.3 Partitioning Datasets

It's often useful to split a dataset into training and testing datasets for most tasks involving machine learning. This can be achieved in SML by using the *SPLIT Keyword*. Figure 5 shows an example of a SML *Query* performing a 80/20 split for training and testing data respectively by utilizing the *SPLIT Keyword* after reading in data.

3.2.4 Using Classification Algorithms

To use a classification algorithm in SML one would use the *CLASSIFY Keyword*. SML has the following classification algorithms implemented: Support Vector Machines, Naive Bayes, Random Forest, Logistic Regression, and K-Nearest Neighbors. Figure 6 demonstrates how to use the *CLASSIFY Keyword* in a *Query*.

3.2.5 Using Clustering Algorithms

Clustering algorithms can be invoked by using the *CLUSTER Keyword*. SML currently has K-Means clustering implemented. Figure 7 demonstrates how to use the *CLUSTER Keyword* in a *Query*.

3.2.6 Using Regression Algorithms

Regression algorithms use the *REGRESS Keyword*. SML currently has the following regression algorithms implemented:Simple Linear Regression, Ridge Regression, Lasso Regression, and Elastic Net Regression. Figure 8 demonstrates how to use the *REGRESS Keyword* in a *Query*.

```
READ "/path/to/data" (separator = ",", header = None)
AND SPLIT (train = 0.8, test = 0.2) AND CLASSIFY
(predictors = [1,2,3,4], label = 5, algorithm = svm)
```

Figure 6: Example using the *CLASSIFY Keyword* in SML. Here we read in data and create training and testing datasets using the *READ* and *SPLIT Keywords* respectively. We then use *CLASSIFY Keyword* with the first 4 columns as features and the 5th column to perform classification using a support vector machine.

```
READ "/path/to/data" (separator = ",", header = None)
AND SPLIT (train = 0.8, test = 0.2) AND CLUSTER
(predictors = [1,2,3,4,5,6,7], algorithm = kmeans)
```

Figure 7: Example using the *CLUSTER Keyword* in SML. Here we read in data and create training and testing datasets using the *READ* and *SPLIT Keywords* respectively. We then use *CLUSTER Keyword* with the first 7 columns as features and perform unsupervised clustering with the K-Means algorithm.

```
READ "/path/to/data" (separator = ",", header = None)
AND SPLIT (train = 0.8, test = 0.2) AND REGRESS
(predictors = [1,2,3,4,5,6,7,8,9], label = 10,
algorithm = ridge)
```

Figure 8: Example using the *REGRESS Keyword* in SML. Here we read in data and create training and testing datasets using the *READ* and *SPLIT Keywords* respectively. We then use *REGRESS Keyword* with the first 9 columns as features and the 10th column to perform regression on using ridge regression.

3.2.7 Saving/Loading Models

It's possible to save models and reuse them later. To save a model in SML one would use the *SAVE Keyword* in a *Query*. To load an existing model from SML one would use the *LOAD Keyword* in a *Query*. Figure 9 shows how the syntax required save and load a model using SML. With any of the existing queries using *REGRESS*, *CLUSTER*, or *CLASSIFY Keywords* attaching *SAVE* to the *Query* will save the model.

3.2.8 Visualizing Datasets and Metrics of Algorithms

When using SML it's possible to visualize datasets or metrics of algorithms (such as learning curves, or ROC curves). To do this the *PLOT Keyword* must be specified in a *Query*. Figure 10 shows can example of how to use the *PLOT Keyword* in a *Query*. We apply the same operations to perform clustering in Figure 7 however we utilize the *PLOT Keyword*.

4. SML's Architecture

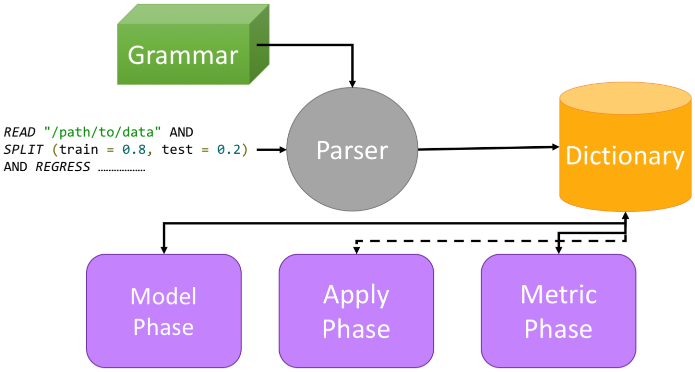
With SML's grammar defined enough information has been presented to dive into SML's architecture. When SML is given a *Query* in the form of a string, it is passed to the parser.

```
SAVE "path/to/save/model"
LOAD "path/to/load/model"
```

Figure 9: Example using the *LOAD* and *SAVE Keywords* in SML.

```
READ "/path/to/data" (separator = ",", header = None)
AND SPLIT (train = 0.8, test = 0.2) AND CLUSTER
(predictors = [1,2,3,4,5,6,7], algorithm = kmeans)
AND PLOT
```

Figure 10: Example using the *PLOT Keyword* in SML.



```
READ "/path/to/data" AND
SPLIT (train = 0.8, test = 0.2)
AND REGRESS ................
```

Figure 11: Block Diagram of SML's Architecture

The high level implementation of the grammar is then used to parse through the string to determine the actions to perform. The actions are stored in a dictionary and given to one of the following phases of SML: Model Phase, Apply Phase, or Metrics Phase. Figure 11 shows a block diagram of this process.

The model phase is generally for constructing a model. The *Keyword*s that generally invoke the model phase are: *READ*, *REPLACE*, *CLASSIFY*, *REGRESS*, *CLUSTER*, and *SAVE*. The apply phase is generally for applying a preexisting model to new data. The *Keyword* that generally invokes the apply phase is *LOAD*. It's often useful to visualize the data that one works with and beneficial to see performance metrics of a machine learning model. By default if you specify the *PLOT Keyword* in a *Query*, SML will execute the metrics phase.

The last significant component of SML's architecture is the connector. The connector connects drivers from different libraries and languages to achieve an action a user wants during a particular phase (see Figure 12). If one considers applying linear regression on a dataset, during the model phase SML calls the connector to retrieve the linear regression library in this case SML uses sci-kit learn's implementation however, if we wanted to use an algorithm not available in sci-kit learn such as a Hidden Markov Model (HMM) SML will use the connector to call another library that supports HMM.

5. Interface

They're multiple interfaces available for working with SML. We've developed a web tool that's publicly available which allows users to write queries and get results back from SML through a web interface (see Figure 13). There's also a REPL environment available that allows the user to interactively write queries and displays results from the appropriate phases

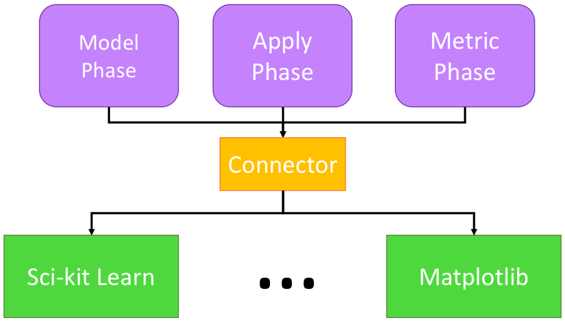
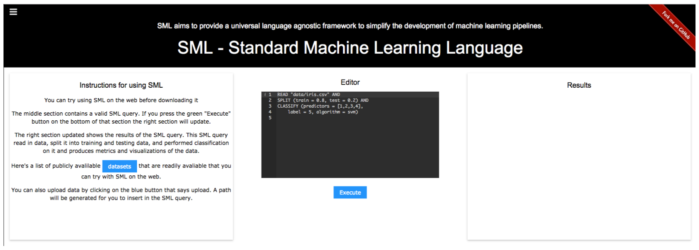


Figure 12: Block Diagram of SML's Connector



SML aims to provide a universal language agnostic framework to simplify the development of machine learning pipelines.

SML - Standard Machine Learning Language

Instructions for using SML

You can try using SML on the web before downloading it

The middle section contains a valid SML query. If you press the green "Execute" button on the bottom of that section the right section will update.

The right section updated shows the results of the SML query. This SML query read in data, split it into training and testing data, and performed classification on it and produces metrics and visualizations of the data.

Here's a list of publicly avalilable datasets that are readily avaliable that you can try with SML on the web.

You can also upload data by clicking on the blue button that says upload. A path will be generated for you to insert in the SML query.

Editor

```
READ "data/iris.csv" AND
SPLIT (train = 0.8, test = 0.2) AND
CLASSIFY (predictors = [1,2,3,4],
    label = 5, algorithm = svm)
```

Results

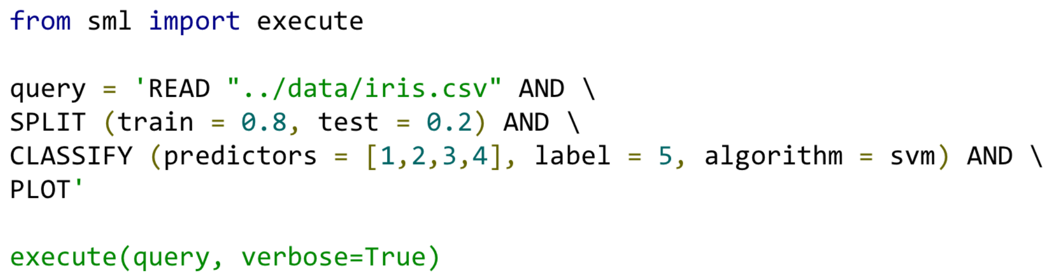
Figure 13: Interface of SML's website. Currently users can read instructions and examples of how to use SML are on the left pane. In the middle pane users can type an SML *Query* and then hit the execute button. The results of running the *Query* through SML are then displayed on the right pane.

of SML. Lastly, users have the option to import SML into an existing pipeline to simplify the development process of apply machine learning to problems.

6. Use Cases

We tested SML's framework against ten popular machine learning problems with publicly available data sets. We applied SML to the following datasets: Iris Dataset [2], Auto-MPG Dataset [3], Seeds Dataset [4], Computer Hardware Dataset [5], Boston Housing Dataset [6], Wine Recognition Dataset [7], US Census Dataset [8], Chronic Kidney Disease [9], Spam Detection [10] which were taken from UCI's Machine Learning Repository (Lichman, 2013). We also

[2] https://archive.ics.uci.edu/ml/datasets/Iris
[3] https://archive.ics.uci.edu/ml/datasets/Auto+MPG
[4] https://archive.ics.uci.edu/ml/datasets/seeds
[5] https://archive.ics.uci.edu/ml/datasets/Computer+Hardware
[6] https://archive.ics.uci.edu/ml/datasets/Housing
[7] https://archive.ics.uci.edu/ml/datasets/Wine
[8] https://archive.ics.uci.edu/ml/datasets/US+Census+Data+(1990)
[9] https://archive.ics.uci.edu/ml/datasets/Chronic_Kidney_Disease
[10] https://archive.ics.uci.edu/ml/datasets/Spambase



```
from sml import execute

query = 'READ "../data/iris.csv" AND \
SPLIT (train = 0.8, test = 0.2) AND \
CLASSIFY (predictors = [1,2,3,4], label = 5, algorithm = svm) AND \
PLOT'

execute(query, verbose=True)
```

Figure 14: SML *Query* that performs classification on the iris dataset using support vector machines. It's important to note that detailed documentation is publicly available in [13] and the purpose of this figure is to highlight the level of the level of complexity relative to an SML query.

applied SML to the Titanic Dataset [11]. In this paper we discuss in detail the process of applying SML to the Iris Dataset and the Auto-MPG dataset [12]. For both of these cases we used the same libraries and programming language in SML and for writing code to solve these use cases.

6.0.1 Iris Dataset

Figure 14 shows all of the code required to perform classification on the Iris dataset using SML in Python. In Figure 14 data is read in from a specified path of a file called "iris.csv" of a subdirectory called "data" in the parent directory, performs a 80/20 split, uses the first 4 columns to predict the 5th column, uses support vector machines as the algorithm to perform classification and finally plot distributions of our dataset and metrics of our algorithm. Appendix A illustrates what is required to perform the same operations using Python and a popular machine learning library scikit learn. The *Query* in Figure 14 and the code in Appendix A use the same 3rd party libraries implicitly or explicitly. It's also worth noting that the code in Appendix A is publicly available and well documented [13] and it is out of the scope of this paper. Instead the complexities required to produce such results with and without SML are outlined. The result for both snippets of code are the same and can be seen in Figure 15.

6.0.2 Auto-Mpg Dataset

Figure 16 shows the SML *Query* required to perform regression on the Auto-MPG dataset in Python. In Figure 16 we read data from a specified path, the dataset is separated by fixed width spaces and we choose not to provide a header for the dataset. Next we perform a 80/20 split, replace all occurrences of "?" with the mode of the column. We then perform linear regression using columns 2-8 to predict the 1st label. Lastly, we visualize distributions

[11] https://www.kaggle.com/c/titanic

[12] Footnote 1 provides detailed explanations and examples that solve problems all 10 data sets

[13] For a detailed documentation describing this code visit: https://github.com/lcdm-uiuc/sml/blob/master/dataflows/plot/iris_svm-READ-SPLIT-CLASSIFY-PLOT.ipynb

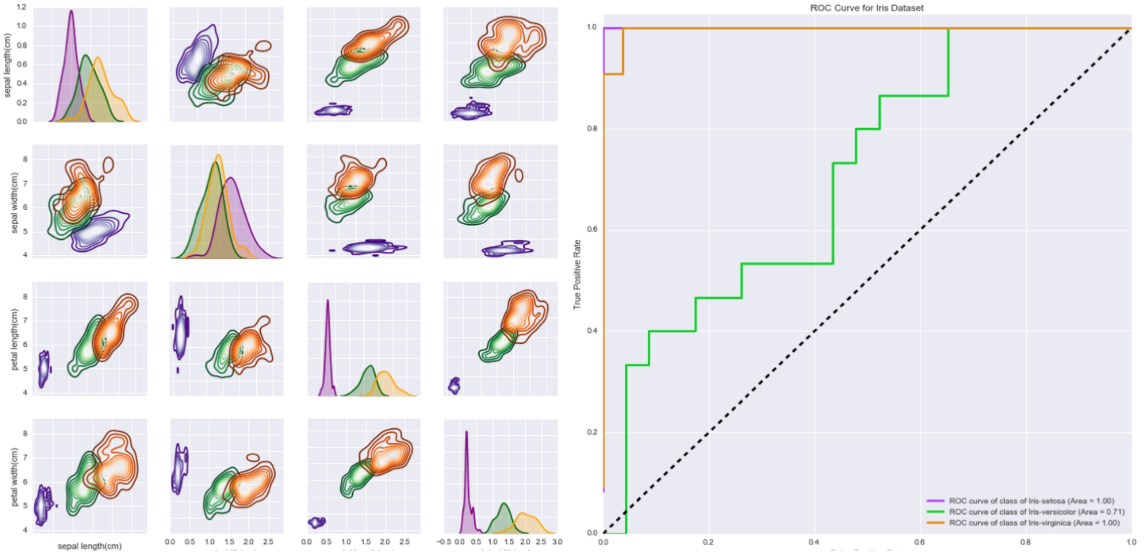
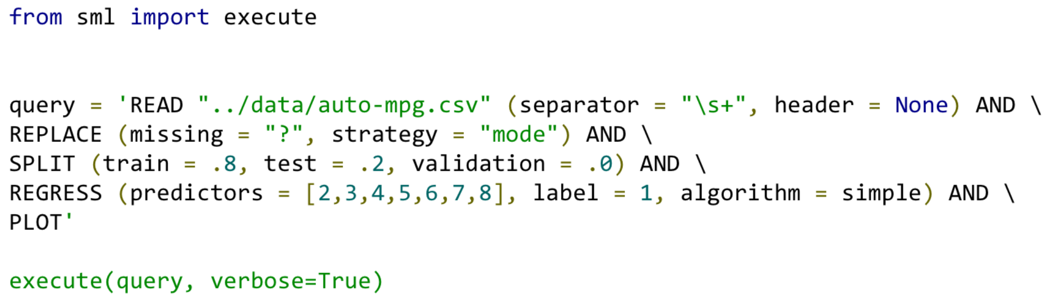


Figure 15: The SML *Query* in Figure 14 and the code in Figure **??** produce these results. The subgraph on the left is a lattice plot showing the density estimates of each feature used. The graph on the right shows the ROC curves for each class of the iris dataset.


```
from sml import execute


query = 'READ "../data/auto-mpg.csv" (separator = "\s+", header = None) AND \
REPLACE (missing = "?", strategy = "mode") AND \
SPLIT (train = .8, test = .2, validation = .0) AND \
REGRESS (predictors = [2,3,4,5,6,7,8], label = 1, algorithm = simple) AND \
PLOT'

execute(query, verbose=True)
```

Figure 16: SML *Query* that performs classification on the Auto-MPG dataset using support vector machines.

of our dataset and metrics of our algorithm. Appendix B. demonstrates what's required to perform the same operations using scikit learn [14]. The outcome of both processes are the same and can be seen in Figure 17.

[14] For a detailed documentation describing this code visit: https://github.com/lcdm-uiuc/sml/blob/master/dataflows/plot/autompg_linear_regression-READ-SPLIT-REGRESS-PLOT.ipynb

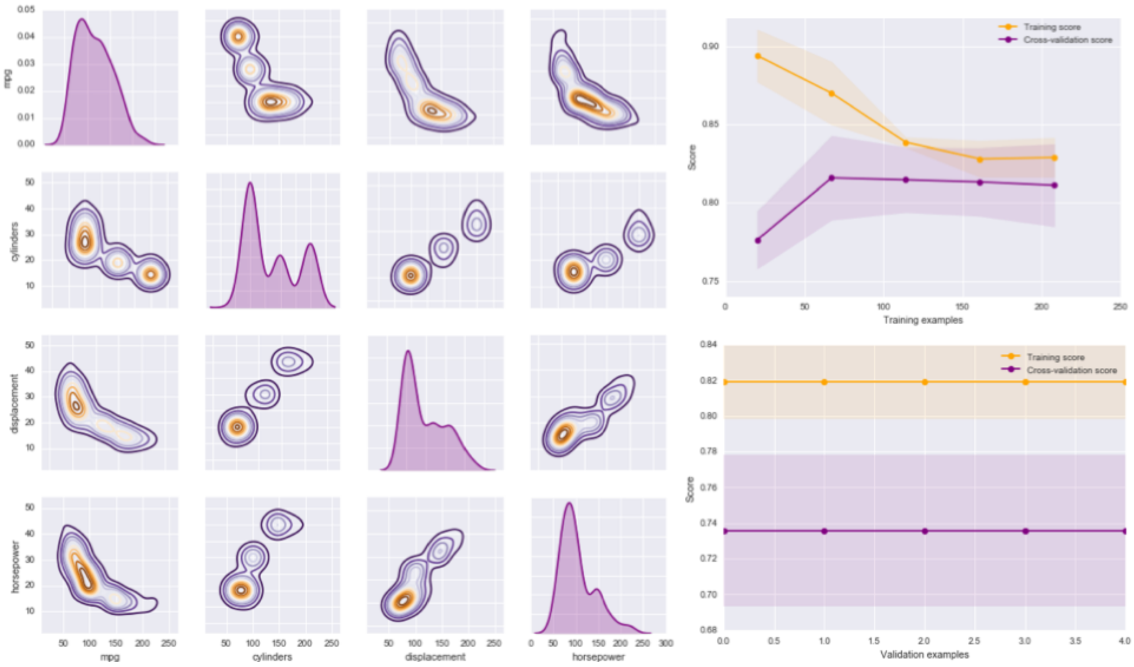


Figure 17: The SML *Query* in Figure 16 and the code in Appendix B. produce these results. The subgraph on the left is a lattice plot showing the density estimates of each feature used. The top right graph shows the learning curve of the model and the graph on lower right shows the validation curve.

6.1 Discussion

For the Iris and Auto-MPG use cases the same libraries and programming language were used to perform regression and classification. The amount of work required to perform a task and produce the following results in Figure 17 and Figure 15 significantly decreases when SML is utilized. Constructing each SML query used less than 10 lines of code however, implementing the same procedures without SML using the same programming language and libraries needed 70+ lines of code. This demonstrates that SML simplifies the development process of solving problems with machine learning and opens a realm of possibility to rapidly develop machine learning pipelines which would be an attractive aspect for researchers.

7. Conclusion

To summarize we introduced an agnostic framework that integrates a query-like language to simplify the development of machine learning pipelines. We provided a high level overview of it's architecture and grammar. We then applied SML to machine learning problems and demonstrated how the complexity of the code one has to write significantly decreases when SML is used. The source code and detailed documentation for SML is open-sourced and publicly available on github [15]. In the future we plan to extend the connector to support more machine learning libraries and additional languages. We also plan to expand the web application to make SML easier to use for a lament user.

If we want researchers from other domain areas to utilize machine learning without understanding the complexities required for machine learning a tool like SML is needed. The concepts presented in this paper as a whole is sound. The details may change but the core principals will remain the same. Abstracting the complexities of machine learning from users is appealing because this will increase the use of machine learning by researchers in different disciplines.

Acknowledgments

We acknowledge support from the National Science Foundation Grant No. AST-1313415. RJB acknowledges support as an Associate within the Center for Advanced Study at the University of Illinois.

Appendix A. Iris Python Code

This shows the code required to replicate the same actions of the SML *Query* in Figure 14. It's important to note that detailed documentation is publicly available in [13], the purpose of this figure is to highlight the level of complexity relative to a SML query.

```python
import pandas as pd
import numpy as np

from sklearn.preprocessing import label_binarize
import sklearn.cross_validation as cv
```

[15] https://github.com/lcdm-uiuc/sml

```
from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import SVC
from sklearn.metrics import roc_curve, auc

import matplotlib.pyplot as plt
import seaborn as sns
names = ['sepal length(cm)', 'sepal width(cm)', 'petal length(cm)', 'petal width(cm)', 'species']
data = pd.read_csv('../data/iris.csv', names=names)

iris_classes = ['Iris-setosa', 'Iris-versicolor', 'Iris-virginica']
features = np.c_[data.drop('species',1).values]
labels = label_binarize(data['species'], classes=iris_classes)
n_classes = labels.shape[1]

x_train, x_test, y_train, y_test = cv.train_test_split(features, labels, test_size=0.25)
svm = OneVsRestClassifier(SVC(kernel='linear', probability=True))
l = svm.fit(x_train, y_train)
predict_score = model.decision_function(x_test)
test_set_results = model.score(x_test, y_test) * 100
print ('SVM Prediction Accuracy = {0:6.2f}%'.format(test_set_results) )
 fpr = dict()
tpr = dict()
roc_auc = dict()

for i in range(n_classes):
fpr[i], tpr[i], _ = roc_curve(y_test[:, i], predict_score[:, i])
roc_auc[i] = auc(fpr[i], tpr[i])
plt.rcParams['figure.figsize']=(12,12)
# Class Info
columns = [0,1,2,3]
cmap_class = ['Purples_r', 'Greens_r', 'Oranges_r', 'Greys_r' ]
color_class1D = ['purple', 'darkgreen', 'orange', 'grey']
column_headers =  data.columns.values.tolist() # Grab headers from df
column_headers = [column_headers[x] for x in columns] # Map headers to indices selected

label = 'species'
fig, ax = plt.subplots(len(columns), len(columns))
for ic, cc, cc1D in zip(iris_classes, cmap_class, color_class1D):
  iris_class_data = data.loc[data.species == ic] # sep class

  #Generate kde plot matrix for class
  for col1, i in enumerate(columns):
      for col2, j in enumerate(columns):
          if i == j:
              sns.kdeplot(iris_class_data[iris_class_data.columns[col1]], ax=ax[col1][col2], color=cc1D, shade=True, legend=False)
          else:
              sns.kdeplot( iris_class_data[iris_class_data.columns[col1]], iris_class_data[iris_class_data.columns[col2]], ax=ax[col1][col2], cmap=cc)
          # Formatting
          if j == 0:
              ax[i,j].set_xticklabels([])
              ax[i,j].set_ylabel(column_headers[i])
              ax[i,j].set_xlabel('')
              if i == len(columns)-1:
                  ax[i,j].set_xlabel(column_headers[j])
          elif i == len(columns)-1:
              ax[i,j].tick_params(axis='y', which='major', bottom='off')
              ax[i,j].set_yticklabels([])
              ax[i,j].set_xlabel(column_headers[j])
              ax[i,j].set_ylabel('')
          else:
              ax[i,j].set_xticklabels([])
              ax[i,j].set_xlabel('')

              ax[i,j].set_yticklabels([])
              ax[i,j].set_ylabel('')

plt.show()
plt.close()
```

Appendix B. Auto-MPG Python Code

This shows the code required to replicate the same actions of the SML *Query* in Figure 16. It's important to note that detailed documentation is publicly available in [14], the purpose of this figure is to highlight the level of complexity relative to a SML query.

```
import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
from sklearn import linear_model
from sklearn.cross_validation import train_test_split
from sklearn.learning_curve import learning_curve, validation_curve

import matplotlib.pyplot as plt
import seaborn as sns

plt.rcParams['figure.figsize']=(12,12)
sns.set()

names = ['mpg', 'cylinders', 'displacement', 'horsepower', 'weight', 'acceleration', 'model_year', 'origin', 'car_name']

#load dataset
data = pd.read_csv('../data/auto-mpg.csv', sep = '\s+', header = None, names = names)
data_clean=data.applymap(lambda x: np.nan if x == '?' else x).dropna()
X = data_clean[['cylinders', 'displacement', 'horsepower', 'weight', 'acceleration', 'model_year', ”origin”]]
#Select target column
y = data_clean['mpg']
#Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.8, test_size=0.2)
```

```

# Define and train  linear regression model
estimator = linear_model.LinearRegression()# Generate Learning Cures
train_sizes, train_scores, test_scores = learning_curve(estimator, X_train, y_train)
# Train Linear Regression Model
estimator.fit(X_train, y_train)# Generate Validation Curves
param_range = np.arange(0, 5)

v_train_scores, v_test_scores = validation_curve(estimator, X_test, y_test, param_name='normalize', param_range=param_range)

score = estimator.score(X_test, y_test)
print('Accuracy␣:', score)
g = sns.PairGrid(data_clean, palette='PuOr_r')
g = g.map_diag(sns.kdeplot, shade=True) # can't add color arg...

g = g.map_upper(sns.kdeplot, cmap='PuOr_r')
g = g.map_lower(sns.kdeplot, cmap='PuOr_r')

plt.show()
plt.close()

color_pal = ['purple', 'dark␣green', 'orange', 'grey'] # For 1-D KDE
cmap_pal = ['PuOr_r'] # For 2-D KDE
classes = [] # May not have a class for categories

column_headers =  data_clean.columns.values.tolist() # Grab headers from df
column_headers = [column_headers[x] for x in columns] # Map headers to indices selected

fig, ax = plt.subplots(len(columns), len(columns))
if not classes:
  for col1, i in enumerate(columns):
      for col2, j in enumerate(columns):
          if i == j:
              sns.kdeplot(data_clean[data_clean.columns[col1]], ax=ax[col1][col2], color=color_pal[0], shade=True, legend=False)
          else:
              sns.kdeplot( data_clean[data_clean.columns[col1]], data_clean[data_clean.columns[col2]], ax=ax[col1][col2], cmap=cmap_pal[0])

           # Formatting
           if j == 0:
               ax[i,j].set_xticklabels([])
               ax[i,j].set_ylabel(column_headers[i])
               ax[i,j].set_xlabel('')
                if i == len(columns)-1:
                   ax[i,j].set_xlabel(column_headers[j])
            elif i == len(columns)-1:
                ax[i,j].tick_params(axis='y', which='major', bottom='off')
                ax[i,j].set_yticklabels([])
                ax[i,j].set_xlabel(column_headers[j])
                ax[i,j].set_ylabel('')
            else:
                ax[i,j].set_xticklabels([])
                ax[i,j].set_xlabel('')
                ax[i,j].set_yticklabels([])
                ax[i,j].set_ylabel('')
plt.show()
plt.close()

plt.figure()
plt.xlabel("Validation examples")
plt.ylabel("Score")

v_train_scores_mean = np.mean(v_train_scores, axis=1)
v_train_scores_std = np.std(v_train_scores, axis=1)
v_test_scores_mean = np.mean(v_test_scores, axis=1)
v_test_scores_std = np.std(v_test_scores, axis=1)

plt.fill_between(param_range, v_train_scores_mean - v_train_scores_std, v_train_scores_mean + v_train_scores_std, alpha=0.1, color="orange")
plt.fill_between(param_range, v_test_scores_mean - v_test_scores_std, v_test_scores_mean + v_test_scores_std, alpha=0.1, color="purple")plt.plot(param_range, v_train_scores_mean, 'o-', color="orange", label="Training score")

plt.plot(param_range, v_test_scores_mean, 'o-', color="purple", label="Cross-validation score")

plt.legend(loc="best")
plt.show()
plt.close()
```

References

Domingos, P. (2012). A few useful things to know about machine learning.. Vol. 55, pp. 78–87, New York, NY, USA. ACM.

Lichman, M. (2013). UCI machine learning repository..

Olson, R. S., Bartley, N., Urbanowicz, R. J., & Moore, J. H. (2016). Evaluation of a tree-based pipeline optimization tool for automating data science. *CoRR*, *abs/1603.06212*.

Rizzolo, N., & Roth, D. (2010). Learning based java for rapid development of nlp systems. In *LREC*, Valletta, Malta.

Roth, D. (2005). Learning based programming. *Innovations in Machine Learning: Theory and Applications*.