Journal of Artificial Intelligence Research ? (YYYY) 1-?? Submitted M/YY; published M/YY

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

Kelechi Ikegwu IKEGWU2@ILLINOIS.EDU
226 Astronomy Building, MC-124
1002 W. Green St.
Urbana, IL 61801

Micheal Hao MXHAO2@ILLINOIS.EDU
ECE Building
306 N Wright St.
Urbana, IL 61801

Robert Brunner BIGDOG@ILLINOIS.EDU
226 Astronomy Building, MC-221
1002 W. Green St.
Urbana, IL 61801

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 it's 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. We then apply SML to a few research problems and compare the complexities of SML queries and traditional approaches used to solve problems. Lastly we perform a case study on SML.

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 solve 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 novice with domain knowledge utilizing machine learning to solve a particular problem 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, 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

```
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.

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

While SML is unique query like language that offers a level of abstraction for writing machine learning code, they're related works that attempt to provide a level of abstraction as well. 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 by automatically creating functions using data driven approaches. LBJava follows the principles of learning based programming by abstracting he details of common machine learning processes. What separates SML from LBJava is that we offer a higher level of abstraction by providing a query like language which allows people who aren't experiences 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:

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

Figure 2: Transcribing Figure 1 into BNF form. Where the top *Query* was defined in Figure 1 and the bottom *Query* is in BNF format.

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

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 $Keywords$ of SML.

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

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

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

Figure 4: Example of an SML *Query* being used.

3.2 Keywords

Currently they're 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*.

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 an *OptionList*. 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 *OptionList* in addition to reading data from the specified path; the *OptionList* 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.

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.

1. Detailed Documentation is 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)
```

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

```
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.

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*.

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.

```
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.

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

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

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. 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 *Keywords* 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 *Keywords* that generally invoke the apply phase are: *LOAD*, and *APPLY*. 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 want

```
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.

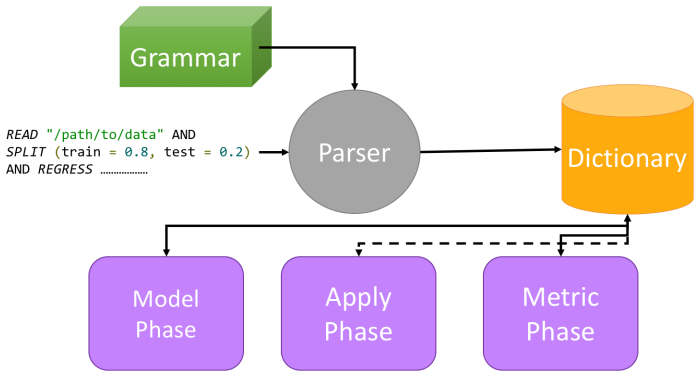

Figure 11: Block Diagram of SML's Architecture

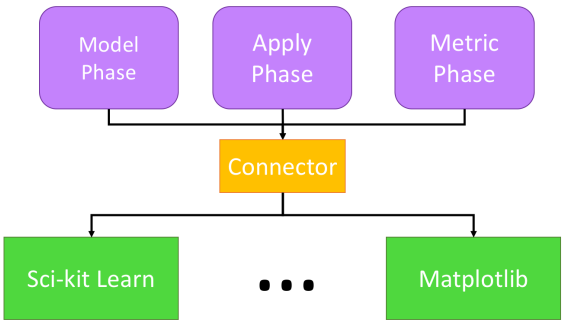

Figure 12: Block Diagram of SML's Connector

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 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 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.

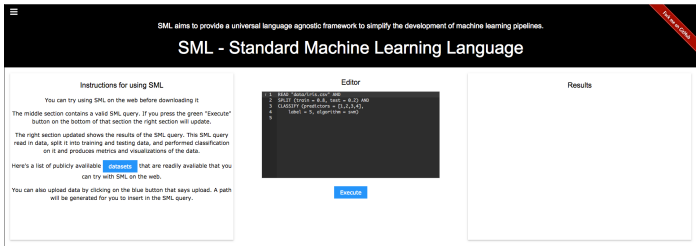

Figure 13: Interface of SML website. The user to interact with SML through a web interface (see Figure 13). Currently users can read instructions on the right to see examples of how to use SML. In the middle pane users can type SML Queries and then hit the execute button. The results of the *Query* are explained on the right pane.

6. Use Cases

We tested the SML 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 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] provides detailed explanations and examples that solve problems all 10 data sets.

6.0.1 Iris Dataset

Figure 14 shows all of the code required to perform classification on the Iris dataset. 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, perform a 80/20 split, use the first 4 columns to predict the 5th column, use support vector machines as the algorithm to perform classification and finally plot distributions of our dataset and metrics of our algorithm. Figure 15 illustrates what is required to perform the same operations using scikit learn. The *Query* in Figure 14 and Figure 15 use the same 3rd party libraries implicitly or explicitly. It's worth noting

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
11. https://www.kaggle.com/c/titanic
12. https://github.com/lcdm-uiuc/sml/tree/master/dataflows

```
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.

```
import pandas as pd
import numpy as np

from sklearn.preprocessing import label_binarize
import sklearn.cross_validation as cv
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))
model = 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()
```

Figure 15: This shows the code required to replicate the same actions of the SML *Query* in Figure 14

that the code in Figure 15 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 16.

6.0.2 Auto-Mpg Dataset

Figure 17 shows the SML *Query* required to perform regression on the Auto-MPG dataset. In Figure 17 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 of our dataset and metrics of our algorithm. Figure 18 demonstrates what's required to perform the same operations using scikit learn. The outcome for both process are the same and can be seen in Figure 19.

13. https://github.com/lcdm-uiuc/sml/blob/master/dataflows/plot/iris_svm-READ-SPLIT-CLASSIFY-PLOT.ipynb

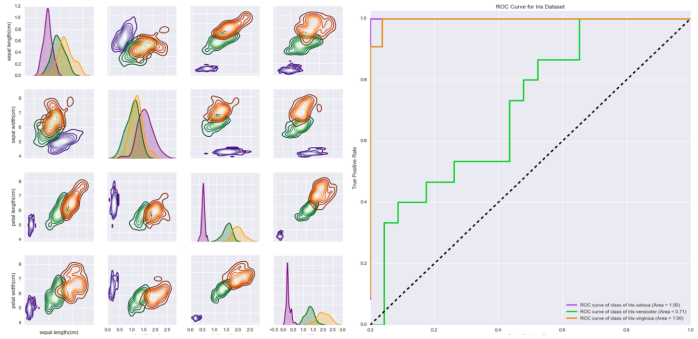

Figure 16: The SML *Query* in Figure 14 and the code in Figure 15 produce these results. The subgraph on the left is a lattice plot showing the density estimates of each feature used. The graph on the right shows the ROC curves for each class of the iris dataset.

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

```
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)
t_train_scores, t_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.c
olumns[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_heade
rs[j])
          elif i == len(columns)-1:
              ax[i,j].tick_params(axis='y',
which='major', bottom='off')
              ax[i,j].set_xticklabels([])
              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_xticklabels([])
              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()
```

Figure 18: This shows the code required to replicate the same actions of the SML *Query* in Figure 17

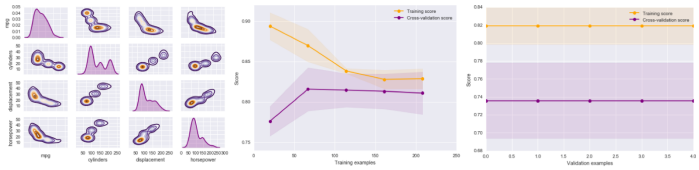

Figure 19: The SML *Query* in Figure 17 and the code in Figure 18 produce these results. The subgraph on the left is a lattice plot showing the density estimates of each feature used. The the centered shows the learning curve of the model and the graph on right shows the validation curve.

6.1 Discussion

For the Iris and Auto-MPG use cases the same libraries were used to perform regression and classification. The amount of work require to perform a task and produce the following results in Figure 19 and Figure 16 significantly decreases when SML is utilized. To construct each of the SML queries less than 10 lines are required however, to implement the same procedures using the same libraries 80+ lines are required. This demonstrates that it 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 [14]. In the future we plan to extend the connector to support more machine learning libraries and additional languages. We plan to expand the web application to make SML easier to use.

If we want to 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 are needed to increase the use of machine learning by researchers in different disciplines.

References

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

14. https://github.com/lcdm-uiuc/sml

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*.