CSCI-513 Project

Machine learning classifiers for calculator expressions

Press down-key, arrow-key to continue.

Introduction

I was intrigued by automatically deducing arbitrary language grammars. But, that is not simple and not a classifier. However, I'm still curious to experiment with the

1. **capabilities** of machine learning in classifying strings within a language grammar,
2. **performance** using exponentially increasing sample sizes, and
3. **behavior** when the set of samples happens to be 100% complete.

Some possible classifiers might be

1. a binary classifier: "correct" versus "error",
2. a multi-class classifier: "correct", versus first "error position", or
3. a multi-class classifier: "correct", versus first "error position" + "error type"

I chose option 2, a **multi-class classifier** with classes of **"correct"** and **"error position"** indicators.

Calculator Expressions

(x * - 1 + 0 / y)

0 and **1**, numbers

x and **y**, memory registers

+ and **-**, add and subtract

***** and **/**, multiply and divide

(and **)**, left and right parenthesis

1. To keep things **simple**, I choose calculator expressions as the language for **testing**.

2. To keep things **simpler**, I choose a **small** set of operand and operator tokens, so only unary and binary addition and subtraction, multiplication, division, and grouping are supported.

3. To keep things the **simplest**, I choose to keep each token exactly **one character** in length. Thus the vocabulary includes the following 10 one-character tokens:

Steps for Classifier Processing

1

Data Acquisition & Ingestion

Gather and **load** dataset into a **Pandas DataFrame**. This may involve reading from a CSV file, database, or even an API. Initially **explore** the data.

2

Data Cleaning & Preprocessing

Clean and preprocess it. This involves handling **missing** or **inconsistent** values, **formatting** data types appropriately, potentially removing **outliers** or **noise**, encoding categorical variables, feature scaling or normalization.

3

Feature Engineering & Label Separation

Clearly **define** your **target** (classification) **column** and **separate** it from the **features**. Split the DataFrame into two parts. Optionally perform feature engineering, creating new features or combining existing ones to better capture patterns in the data.

4

Data Splitting

Partition your dataset into **training** and **testing** sets, and occasionally include a validation set. The typical ratio might be **70–80%** for training and **20–30%** for testing, but this depends on your dataset size and problem specifics.

Steps for Classifier Processing

5

Model Selection & Fitting

Choose a suitable classifier based on the problem and data characteristics. Instantiate the model and **fit (train)** it using the training data. This step involves the model learning the underlying patterns in your features relative to the target.

6

Prediction

Use the model to make predictions on the **test set**. This step is where you see how well the learned patterns generalize to unseen data.

7

Evaluation

Assess the **performance** of the classifier using appropriate metrics like **accuracy**, **precision**, **recall**, **F1-score**, and a **confusion matrix**, among others. In imbalanced datasets, precision and recall could be more telling than plain accuracy.

8

Hyperparameter Tuning & Validation

Optionally, steps 5–7 are often **iterated** over with hyperparameter tuning to **refine** the model's **performance**. This step ensures that your chosen parameters are optimized for the data, reducing the risk of underfitting or overfitting.

Data

The calculator expressions are **stored in tables**. The target column contains the class; **zero (0)** indicates **correct**, and **positive integers** indicate the **character position** of the **first error**. Each table stores the set of expressions for a given fixed length with each column containing a one token-character ordinal. The column names are target, x1, x2, x3, etc.

The tables are **generated dynamically**. All possible expressions for the given length are generated exhaustively, and the target value for each is calculated via a blazingly-fast hand-written top-down recursive-descent parser using the Cython package.

So, the first table has **10** rows, the second has **100** rows, **1_000**, **10_000**, **100_000**, etc.

Models

The data features are simply the tokens and their implicit positions. I compare and contrast these five models from SciKit-learn:

1. Logistic Regression, max_iter=200
2. Multinomial NB
3. Support Vector Classifier
4. Random Forest Classifier, n_estimators=100
5. Decision Tree Classifier

As a bonus and just for fun, but not part of this project, I also compared these five models from PyTorch:

1. Transformer-based Model using a single encoder layer
2. Feedforward network with one hidden layer
3. Deeper network with two hidden layers
4. RNN using an LSTM in which the hidden state of the last time step is used for classification
5. Convolutional Neural Network with 1D Convolutions

1. Steps for Classifier Processing
2. Compare SciKit-learn performance
3. Compare PyTorch performance

Agenda

1 Data Acquisition

To exhaustively generate calculator expressions of a given length, I use the product function from itertools module. Check out the powerful yield statement for generating values. Some of the models are unable to handle non-numeric data so an ordinal value is used instead.

```
from itertools import product
total = None
tokens = "(x*-1+0/y)"
def expressions(length):
    global total; total = 1
    for toks in product(tokens, repeat=length):
        expression = "".join(toks)
        yield (parse_expression(expression), expression)
        total += 1
    total -= 1
```

```
token_map = {'(': 1, 'x': 2, '0': 3, '+': 4, '*': 5, '/': 6, '-': 7, '1': 8, 'y': 9, ')': 10}
def EXPRESSIONS(length):
    XS = []
    for expr in expressions(length):
        XS.append([expr[0]] + [token_map[x] for x in expr[1]])
    return XS
```

1 Exploratory Data Analysis


```
for length in range(1, 8):
    start = time.time_ns() // 1000
    classes = [0] * (length+2)
    for expr in expressions(length):
        classes[expr[0]] += 1
    finish = time.time_ns() // 1000
    classes = [(c * 100.0 / total) for c in classes]
    pprint([length, total, (finish - start) / 1000000.0, classes])
```

Test generating and classifying expressions from length 1 to length 7. Using Cython, the loop generates over 10 million expressions in about 20 seconds.

1 Data Acquisition

To generate the target classification, a very simple parser was built in Python. However, it was too slow, so I utilized the Cython package for an almost 200-times speed increase.

```
!pip install Cython
%load_ext cython
```

```
%%cython

cdef int pos
cdef bytes subject

cdef void factor():
    global pos, subject
    if   subject[pos] == ord('+'): pos += 1; factor()
    elif subject[pos] == ord('-'): pos += 1; factor()
    elif subject[pos] == ord('('):
        pos += 1
        expr()
        if subject[pos] == ord(')'): pos += 1
        else: raise Exception(pos)
    elif subject[pos] == ord('x'): pos += 1
    elif subject[pos] == ord('y'): pos += 1
    elif subject[pos] == ord('0'): pos += 1
    elif subject[pos] == ord('1'): pos += 1
    else: raise Exception(pos)
```

```
cdef void term():
    global pos, subject
    factor()
    while (  subject[pos] == ord('*')
          or subject[pos] == ord('/')
          ):
          pos += 1
          term()

cdef void expr():
    global pos, subject
    term()
    while (  subject[pos] == ord('+')
          or subject[pos] == ord('-')
          ):
          pos += 1
          expr()
```

```
cdef void stmt():
    global pos, subject
    expr()
    if subject[pos] != ord('\n'):
        raise Exception(pos)

def parse_expression(s):
    global pos; pos = 0
    global subject; subject = f"{s}\n".encode('ascii')
    try: stmt()
    except Exception as e:
        return e.args[0]+1
    return 0
```

3 Label Seperation 4 Data Splitting 5 Model Selection


```
results = {}
warnings.filterwarnings("ignore")
for length in range(1, 6):
    columns = ['target'] + [f"x{i}" for i in range(1, length+1)]
    data = pd.DataFrame(EXPRESSIONS(length), columns=columns)
    X = data.iloc[:, 1:length+1]
    y = data.iloc[:, 0]
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.3, random_state=53, stratify=y
    )
    models = {
        "Logistic Regression": LogisticRegression(max_iter=200),
        "MultinomialNB": MultinomialNB(),
        "Support Vector Classifier": SVC(),
        "Random Forest Classifier": RandomForestClassifier(n_estimators=100),
        "Decision Tree Classifier": DecisionTreeClassifier()
    }
```

Keeping results for all runs, iterate for each length from 1 to 5.

5 Fitting

6 Prediction

7 Evaluation


```
results[length] = {}
for model_name, model in models.items():
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    results[length][model_name] = {
        "accuracy": accuracy_score(y_test, y_pred),
        "precision": precision_score(y_test, y_pred, average='weighted'),
        "recall": recall_score(y_test, y_pred, average='weighted'),
        "F1": f1_score(y_test, y_pred, average='weighted')
    }
    print(f"sklearn: {length}. {model_name}")
    print(classification_report(y_test, y_pred))
```

Iterate for each of the five models. Do fit, predict, and gather metrics.

7

Evaluation

Create a table for presenting the comparison results. Use Python's dictionary construction to populate. Use Pandas T property for transposing the table in like manner as transposing a matrix.

```
comparison = pd.DataFrame(
{ model_name: {
            "accuracy": metrics["accuracy"],
            "precision": metrics["precision"],
            "recall": metrics["recall"],
            "F1": metrics["F1"]
    } for model_name, metrics in results[length].items()
}).T
print(f"sklearn: {length}. Comparison")
print(comparison)
print()
```

SciKit-learn comparison chart (L=1, 10 samples)

	Accuracy	Precision	Recall	F1
Logistic Regression	0.000000	0.000000	0.000000	0.000000
MultinomialNB	0.333333	0.111111	0.333333	0.166667
Support Vector	0.333333	0.111111	0.333333	0.166667
Random Forest	**0.666667**	**0.500000**	**0.666667**	**0.555556**
Decision Tree	**0.666667**	**0.500000**	**0.666667**	**0.555556**

SciKit-learn comparison chart (L=2, 100 samples)

	Accuracy	Precision	Recall	F1
Logistic Regression	0.466667	0.430128	0.466667	0.441414
MultinomialNB	0.366667	0.226488	0.366667	0.279111
Support Vector	0.466667	0.431250	0.466667	0.423556
Random Forest	0.800000	0.779848	0.800000	0.783470
Decision Tree	**0.866667**	**0.888889**	**0.866667**	**0.875789**

SciKit-learn comparison chart (L=3, 1000 samples)

	Accuracy	Precision	Recall	F1
Logistic Regression	0.296667	0.270808	0.296667	0.250282
MultinomialNB	0.373333	0.295459	0.373333	0.310078
Support Vector	0.486667	0.406154	0.486667	0.390813
Random Forest	0.876667	0.876188	0.876667	0.874543
Decision Tree	**0.923333**	**0.923942**	**0.923333**	**0.923556**

SciKit-learn comparison chart (L=4, 10000 samples)

	Accuracy	Precision	Recall	F1
Logistic Regression	0.304000	0.204984	0.304000	0.233935
MultinomialNB	0.387000	0.306045	0.387000	0.308587
Support Vector	0.603000	0.533570	0.603000	0.478860
Random Forest	**0.984000**	**0.984418**	**0.984000**	**0.983951**
Decision Tree	0.975667	0.976144	0.975667	0.975703

SciKit-learn comparison chart (L=5, 100000 samples)

	Accuracy	Precision	Recall	F1
Logistic Regression	0.315067	0.223300	0.315067	0.244833
MultinomialNB	0.372633	0.293235	0.372633	0.305641
Support Vector	0.685033	0.646947	0.685033	0.618648
Random Forest	0.992500	0.992533	0.992500	0.992497
Decision Tree	**0.993133**	**0.993166**	**0.993133**	**0.993143**

PyTorch comparison chart (L=2, 100 samples)

	Accuracy	Precision	Recall	F1
Transformer	**0.60**	**0.540000**	**0.60**	**0.566667**
Feedforward NN	0.45	0.421429	0.45	0.404848
Deeper NN	0.55	0.529286	0.55	0.515385
RNN (LSTM)	0.50	0.527143	0.50	0.490310
CNN	0.35	0.240000	0.35	0.284444

PyTorch comparison chart (L=3, 1000 samples)

	Accuracy	Precision	Recall	F1
Transformer	0.860	0.856150	0.860	0.854777
Feedforward NN	0.895	0.896275	0.895	0.893379
Deeper NN	0.930	0.933831	0.930	0.931149
RNN (LSTM)	**0.940**	**0.945993**	**0.940**	**0.939405**
CNN	0.880	0.886432	0.880	0.871591

PyTorch comparison chart (L=4, 10000 samples)

	Accuracy	Precision	Recall	F1
Transformer	0.989	0.989384	0.989	0.988958
Feedforward NN	**1.000**	**1.000000**	**1.000**	**1.000000**
Deeper NN	**1.000**	**1.000000**	**1.000**	**1.000000**
RNN (LSTM)	**1.000**	**1.000000**	**1.000**	**1.000000**
CNN	**1.000**	**1.000000**	**1.000**	**1.000000**

PyTorch comparison chart (L=5, 100000 samples)

	Accuracy	Precision	Recall	F1
Transformer	0.99970	0.999700	0.99970	0.999700
Feedforward NN	**1.00000**	**1.000000**	**1.00000**	**1.000000**
Deeper NN	**1.00000**	**1.000000**	**1.00000**	**1.000000**
RNN (LSTM)	**1.00000**	**1.000000**	**1.00000**	**1.000000**
CNN	**1.00000**	**1.000000**	**1.00000**	**1.000000**

Conclusion & Future Directions

1

Takeaway 1

Some SciKit-learn classifiers reached mid-80% metrics with only 100 samples, over 90% with 1000 samples, and 98% with 10_000 samples.

2

Takeaway 2

Most of the PyTorch neural networks all reached 100% metrics with 10_000 and 100_000 samples.

And the Transformer was close behind with 0.98 and 0.99 metrics.

3

Takeaway 3

1st: Random Forest. Decision Tree classifiers.

2nd: Support Vector Classifier.

3rd: Logistic Regression and Multinomial NB classifier.

4

Takeaway 4

Executing tests for expressions of length one wasn’t the most useful test.

However, I was a little curious about it.

5

Future Directions

Continuing with the exhaustive approach which will require utilizing the GPU on a local machine.

Generating random samples for the larger lengths.

The End

lcherryh@yahoo.com

Thank you!