Continuous detoxification -- GSOC 2018 Proposal

Organization: NumFocus -- Shogun

- Name:
- GitHub:
- Email:
- Timezone:
- IRC:

Abstract

Shogun is a powerful machine learning toolkit. The project has a long history and a huge codebase. Some parts are very outdated and not well-designed. Polishing the codebase and bringing Shogun to modern design will make it much easier to developers, and as such make the project more attractive for scientists to implement their work in. This GSoC project aims at re-designing Shogun's data representation and some APIs, including features, labels and preprocessors, and bringing novel un-templated data classes with support for lazy evaluation to Shogun. By the end of this project, we expect an improvement of maintainability, stability, and beauty to the codebase of Shogun.

Project Proposal

The primary goal is to make features immutable. There are many bad things, i.e. those that modify underlying data of features directly, which need to be either refactored or dropped. We will split it into several steps.

First, we need to refactor existing algorithm implementations that rely on mutable features. For example, some use the old iterator APIs that expose underlying data using raw pointers, some modify feature matrix directly. After we get rid of them, we can safely transit to immutable features. And then, we can have views of features and labels that share the underlying data. Once we have immutable features, we can enable efficient parallel cross validation by sharing features.

Untemplated matrix and vector are another point to address this summer. There is a prototype that wrap the data and use a switch statement to check the primitive type and then dispatch at runtime. This is inefficient because we have to use the switch every time. Therefore, we will continue from the ongoing PR, and bring support for lazy evaluation using expression template. After we have untemplated data types, we can implement untemplated features and refactor the existing code. I will leave this part as future work after this GSoC.

Preprocessor API

Preprocessor APIs need redesign. Instead of applying to features in-place, which breaks immutability, we will use preprocessor as transformer. And then we can drop the whole preprocessor stuff from *CFeatures*.

```
 // initialize and set up parameters
(constructor)

// fit into training features, preprocessors that do not require
// training data have a empty implementation
void fit(CFeatures *);

// apply to features by creating a new instance
Some<CFeatures *> transform(CFeatures *);
```

View

The view method call creates a new instance of feature or label, which shadow-copies underlying data. A subset is added to the subset stack of the new instance. After this, we will make all subset APIs in CFeatures and CLabels private.

```
Some<CFeatures> CFeatures::view(SGVector<index_t> index);
Some<CLabels> CLabels::view(SGVector<index_t> index);
```

Taken from @micmn 's design, we need to solve the issue of covariant type.

```
// create a new instance
@non-virtual
Some<Features> Features::view(SGVector<index_t> idx);

// do the type cast
@non-virtual
Some<DenseFeatures<T>> DenseFeatures::view(SGVector<index_t> idx)
    auto feats = wrap(
        static_cast<DenseFeatures<T>*>(
            Features::view(idx).get()))
    return feats
```

Feature Iterator

We would like to access features via iterators. Last summer's project brought *DotIterator*. However, there are a set of old iteration APIs in *CDotFeatures*, e.g.

```
void* get_feature_iterator(int32_t vector_index);
bool get_next_feature(int32_t& index, float64_t& value, void* iterator);
void free_feature_iterator(void* iterator);
```

These APIs expose data as raw pointers. We will adapt them to new iterator design in *DotIterator*. This also involves refactor in LibLinear where they are mostly used.

Refactor non-const methods of features

We could start from *CDotFeatures*, which is the super class of many other feature types. There are many non-const methods, for example,

```
virtual float64_t dot(int32_t vec_idx1, CDotFeatures* df, int32_t
vec_idx2)=0;
virtual void add_to_dense_vec(float64_t alpha, int32_t vec_idx1, float64_t*
vec2, int32_t vec2_len, bool abs_val=false)=0;
```

They are non-const because it will call other non-const methods to get the actual feature vector, i.e. get_feature_vector(int32_t num, int32_t& len, bool& dofree) in CDenseFeatures case, which may compute feature vector on the fly (using implementation of subclasses) and then cache it. We can make cache mutable so that these methods can be const.

Parallel Cross Validation

When we have immutable features, we can easily enable parallel cross validation. First we need a suite of tests to systemically test cross validation of each type of *CMachine*. The tests can be generated with a Jinja template, like serialization tests. The test will run cross validation in both single thread and multi-thread mode, and expect that they yield the same result.

We also need to take care of thread safety. For example the *CCache* in *CDenseFeatures* is not thread safe. This can be addressed by adding locks. And then we are able to enable parallelization.

Untemplated Matrix and Vector

We will have *Matrix* and *Vector* as a simple wrapper of templated data. We use expression template for lazy evaluation.

Exp is the base class of all expressions. All expressions should support two evaluation functions, *eval* and *eval_templated*.

```
template <class Derived>
class Exp {
public:
	Derived& self();
	const Derived& self() const;

	typename Derived::untemplated_result_type eval();

	template <PType>
	typename Derived::result_type<PType> eval_templated();
};
```

It has the following direct subtypes that define the corresponding *result_type* and *untemplated_result_type*.

```
// MatrixExp represents a expression that evaluates to a Matrix
template <class Derived>
class MatrixExp: public Exp<MatrixExp<Derived>> {
public:
	typedef Matrix untemplated_result_type;

	template <PType>
	typedef SGMatrix<PType> result_type;
};

// ScalarExp and VectorExp can be defined similarly
template <class Derived>
class VectorExp;

// untemplated_result_type and result_type are both primitive type in
// ScalarExp
template <class Derived>
class ScalarExp;
```

We have several expression types, e.g.

```
template<class OP, class E> class UnaryMatrixExp;
template<class OP, class E> class UnaryVectorExp;
template<class OP, class E1, class E2> class BinaryMatrixExp;
template<class OP, class E1, class E2> class BinaryVectorExp;
```

Matrix can be implicit cast to *UnaryMatrixExp* that evaluates to itself, so as to *Vector*.

A possible implementation is,

```
template<class OP, class E1, class E2>
class BinaryMatrixExp: public MatrixExp<BinaryMatrixExp<OP, E1, E2>>  {
public:
      BinaryMatrixExp(const Exp<E1>& left, const Exp<E2>& right);

      template<class PType>
      result_type<PType> eval_templated() {
            return OP::apply<PType>(
                  e1.eval_templated<PType>(),
                  e2.eval_templated<PType>()
            );

      untemplated_result_type eval() {
            switch(ptype()) {
            case FLOAT64:
                  return untemplated_result_type(
                        eval_templated<float64_t>())
                  );
            ...
            }
      }

private:
      const Exp<E1>& left;
      const Exp<E2>& right;
};
```

We need to wrap linalg functions as static methods. All OP classes have a static method *apply* that takes templated data as input.

```
struct MatrixAdd {
      typedef Matrix result_type;
      template<class T>
      static SGMatrix<T> apply(
            const SGMatrix<T>&,
            const SGMatrix<T>&
      );
};
```

And then, we need to provide factory methods to construct exp depending on the result type of operations and number of arguments. We will define several overloaded versions that return different types of expressions, e.g

```
template<OP, E1, E2,
class = typename enable_if<is_same<Matrix, typename OP::result_type>::value
>::type>
BinaryMatrixExp makeExp(const Exp<E1>&, const Exp<E2>&);
```

Finally, we can define linalg APIs over expressions. For example, we can overload operators like

```
template<E>
auto operator+(const MatrixExp<E>& left, const MatrixExp<E>& right) {
      return makeExp<MatrixAdd>(left, right);
}
```

The lazy expression supports implicit evaluation using the assignment operator.

```
template<class T, class E>
T& operator(T& target, const Exp<E>& exp) {
      target = exp.eval();
      return target;
}
```

A example usage:

```
Matrix X;
Vector w, b;
// initialize these variable in some ways ...

BinaryMatrixExp wx_exp = w * X;
BinaryMatrixExp Y_exp = wx_exp + Y;
Matrix Y = Y_exp; // implicit evaluation
// or trigger evaluation explicitly
Y = Y_exp.eval();
```

Split implementation from header files

Many parts of Shogun are split into .cpp and .h, which makes compilation/development much easier: changes in the implementation of a low level data-structure does not cause the whole project to be re-compiled. There are many cases, however, where this is not done (especially in templated code).

Timeline

Community Bonding

- Dive deeper into the codebase.
- Discuss with the mentor and the community to elaborate the design.
- Split implementation from header files. This task is very easy and can serve as good initial work.

Week 1

- Adapt old feature iterator APIs to DotIterator.
- Refactor usage of old feature iterator in algorithm implementations (mostly LibLinear). #4169 #4175

Week 2 - 3

- Refactor preprocessor and implement new preprocessor APIs.
- Use new preprocessor APIs in existing code.

Week 4 - 5

- Get rid of modification of feature vector or matrix in existing code.

Milestone (First Evaluation)

- By this time, we will have the preprocessor and feature iterators redesigned and refactored.

Week 6

- Implement view of features and labels (almost done by the ongoing PR #3970)
- Use views in existing code (This involves a lot of refactor in codebase, including decision tree algorithms, cookbook, examples and some other algorithms) and remove subset things. #2136

Week 7

- Refactor other non-const methods of CFeatures. #2530 #3781
- If features are immutable, we can then eliminate unnecessary copy in other code.

Week 8

- Add tests of cross validation. #3732

Milestone (Second Evaluation)

- By this time, we will get rid of most non-const methods in CFeatures and it will be nearly immutable. Besides, parallel cross validation are supported.

Week 9

- Support parallel cross validation. #3688 #3743

Week 10

- Implement un-templated matrix and vector. #3782
- Implement expression types for lazy evaluation.

Week 11 - 12

- Create operation types for un-templated data.
- Add linalg APIs for expression types.
- Clean up and finish documents. Get prepared for final evaluation.

About Me

I am a junior student major in Software Engineering in Shanghai Jiao Tong University. I am interested in dealing with complex codebase and enthusiastic in open source. I got involved with Shogun in January this year and have been working on refactoring and replacing old code with new linalg framework [1]. In addition, several new APIs have been added to linalg.

Before that, I worked on several C++ projects, including programming with OpenGL, Unix filesystem. More project experiences include developing websites (Javascript, Java), designing and implementing computer vision and deep learning related algorithms and experiments (Python, C++, Matlab), building tiny operating system (xv6, MIT 6.828 labs).

Besides, I am currently a research assistant in Machine Vision Intelligence Lab [3] and have experience with machine learning and deep learning. I am familiar with other frameworks such as scikit-learn, MxNet, PyTorch and have tried diving deep into these codebase, both in C++ and Python. I believe these experiences would made be capable for this project. My résumé is available at [2] for more information.

Other commitment

The school semester ends at June 15. The courses will require about 10 hrs/ week. In the first few weeks of GSoC, I will work on my spare time after classes so that I will be able to work 40 hours a week. Alternatively I can start working 2 weeks earlier if necessary. After the school semester, I will work full time the whole summer.

Links

[1] Accepted PRs to Shogun
https://github.com/shogun-toolbox/shogun/pulls?q=is%3Apr+author%3Avinx13
[2] Resume [dropped]
[3] Homepage of MVIG http://www.mvig.org/
[4] Project Idea https://github.com/shogun-toolbox/shogun/wiki/GSoC_2018_project_detox