A likelihood discriminator: discussion

P. Ronchese
Dipartimento di Fisica e Astronomia “G.Galilei”

Università di Padova

“Object oriented programming and C++” course

The simple solution

The solution shown as “Likelihood discriminator - version 01” works fine, but some weakness may be found in it:

- the produced `class` (i.e. `EventDiscriminator`) is tightly connected to the specific problem it addresses,
- it cannot be used in other environments, despite the basic solution is very general (classify events on probabilty base),
- a lot of modifications are needed to implement an analogous procedure to classify objects with different nature.

Specific points

Should be necessary to produce a similar `class`
a lot of changes are to be done

- In `EventDiscriminator.h` :
  - the object containing data (`Event`)
    in `fill` and `get` functions,
  - the list of variables,
  - the list of histograms.
- In `EventDiscriminator.cc` :
  - the sequence of histograms creation in `book`,
  - the sequence of histograms filling in `fill`,
  - the sequence of histograms normalization and
    saving to file in `save`,
  - the sequence of histograms reading from file in `read`,
  - the sequence of probability calculations in `get`,
  - the variables calculation.

Many of them do actually count twice (signal+background).

Common and specific operations

A lot of operations are actually
common to any likelihood discriminator

The following operations are to be done independently on the actual variable/histogram list:

- histograms creation,
- histograms filling,
- histograms normalization and saving to file,
- histograms reading from file,
- probability calculations.

Only a rather small part of the operations is actually
specific to the actual problem

- the definition of the variables list,
- their calculation from the object containing data.

Generic and specific tasks separation

Inheritance may help in keeping distinct common and specific operations

- The base `class` contains the common operations and:
  - a “generic” list of variables and histograms,
  - some functionality to accept actual variables for a specific problem.
- The derived class contains the specific operations and:
  - instructions to pass the specific variables to the base `class`,
  - functions to call the base `class` functions according to the specific needs.

Train and test tasks separation

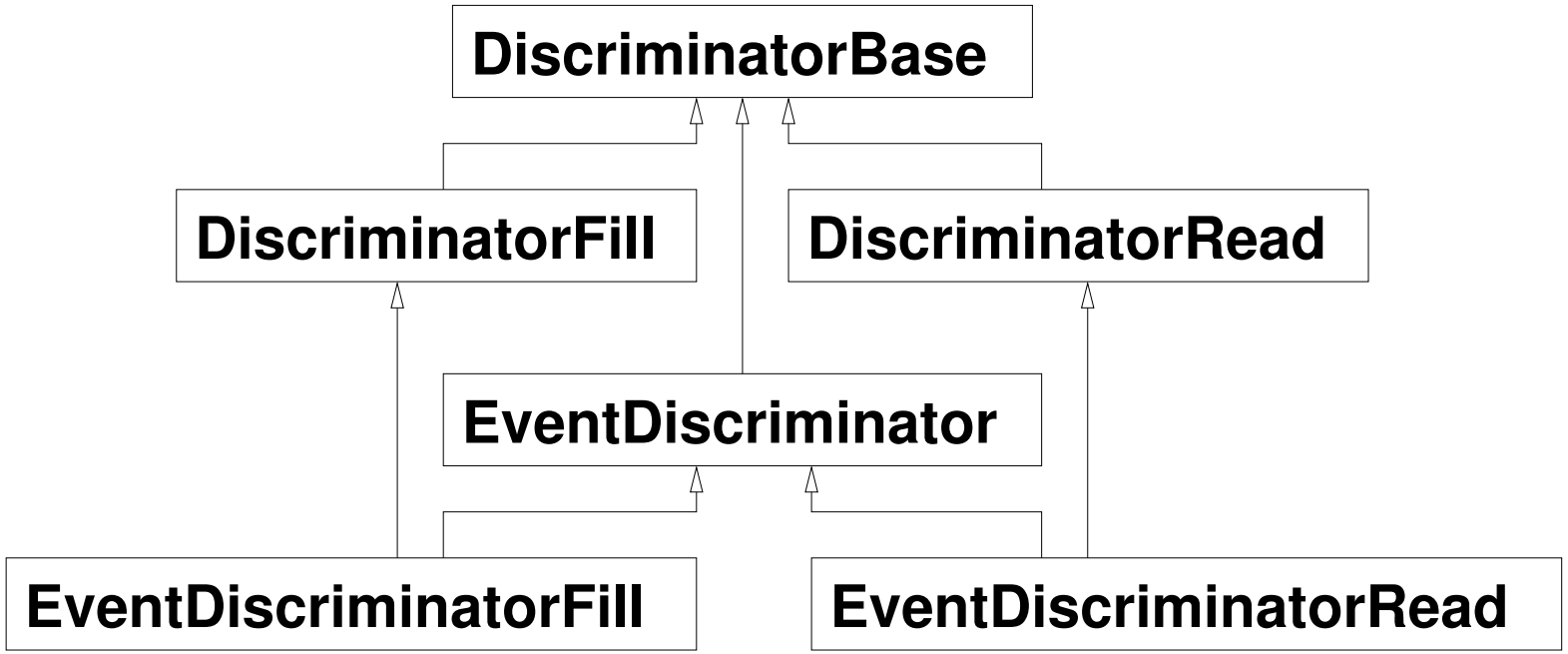
The likelihood discriminator calculation is actually split in two parts

- "Training":
  - already classified events are read,
  - histograms are created,
  - histograms are filled,
  - histograms are normalized and written to file.
- "Testing":
  - histograms are read from file,
  - unknown events are read,
  - probabilities and discriminator are computed.

The "histogram writing" and "histogram reading" parts can be splitted from the "core" part handling the list of variables, both for the generic and specific `class`es.

Object relations

The whole system can be splitted in 6 `class`es



The structure is a bit complicated, but it can be very flexible

Generic variables and histograms list

For each quantity several objects are needed, and the base `class` (i.e. `DiscriminatorBase`) should define a `struct varDesc` to contain all of them:

- a variable to contain its value,
- an histogram with its distribution for signal,
- an histogram with its distribution for background,
- a flag to decide about using it or not.

They should be conveniently labelled with a name, useful to create and store the histograms and to choose at runtime which quantities to use.

A `std::map<std::string,varDesc*>` is the best-suited object to contain the information about the actual quantities used for the specific problem

Variable handling

A function to “register” actual variables in the list is necessary

```
class DiscriminatorBase {
  ...
  struct varDesc {
    bool use;
    float* content;
    TH1F* sigHisto;
    TH1F* bkgHisto;
  };
  void registerVar(const std::string& name,
       float* content,bool active=true);
  ...
  std::map<std::string,varDesc*> varMap;
  ...
};
```

Variable list filling

```
void DiscriminatorBase::registerVar(
     const string& name,float* content,
     bool active) {
  varDesc* vd=new varDesc;
  vd->use     =active;
  vd->content =content;
  vd->bkgHisto=
  vd->sigHisto=0;
  varMap[name]=vd;
  return;
}
```

Specific variable registration

All variables declared in the derived `class` have to be "registered" in the base `class`

```
class EventDiscriminator:
      public virtual DiscriminatorBase {
  ...
  EventDiscriminator();
  ...
  float ptSum;
  float ptMin;
  ...
};
```

```
EventDiscriminator::EventDiscriminator() {
  registerVar("ptSum",&ptSum,true);
  ...
}
```

Looping functions

A lot of functions can simply loop over variables/histograms

```
float DiscriminatorBase::get() const {
  float dSig=1.0;
  float dBkg=1.0;
  map<string,varDesc*>::const_iterator
                          iter=varMap.begin();
  map<string,varDesc*>::const_iterator
                          iter=varMap.end();
  while(iter!=iend) {
    varDesc* vd=(*iter++)->second;
    if (!vd->use ) continue;
    get(*vd->content,dSig,dBkg,
        vd->sigHisto,vd->bkgHisto );
  }
}
```

Histogram declaration

The only other function to be called for each specific quantity is the histogram booking

```
class DiscriminatorFill:
      public virtual DiscriminatorBase {
  ...
  void book(const std::string& name,
            int nb,float xmin,float xmax);
  ...
};
```

Generic histogram creation

```
void DiscriminatorFill::book(
      const string& name,
      int nb,float xmin,float xmax) {
  map<string,varDescription*>::const_iterator
      iter=varMap.find( name );
  map<string,varDescription*>::const_iterator
      iend= varMap.end();
  if(iter==iend) return;
  varDescription* vd=iter->second;
  string sName="hSig_"+name;
  string bName="hBkg_"+name;
  vd->sigHisto=new TH1F(sName,nb,xmin,xmax);
  vd->bkgHisto=new TH1F(bName,nb,xmin,xmax);
  return;
}
```

Specific histogram creation

```
class EventDiscriminatorFill:
      public DiscriminatorFill,
      public EventDiscriminator {
  ...
  void book();
  ...
};
```

```
void EventDiscriminatorFill::book() {
  DiscriminatorFill::book("ptSum",
                          20,0.0,100.0);
  ...
}
```

Discriminator calculation

The specific discriminator variable can be easily computed by calling the generic function

```
class EventDiscriminatorRead:
      public DiscriminatorRead,
      public EventDiscriminator {
  ...
  float get( const Event* ev );
  ...
};
```

```
float EventDiscriminatorRead::get(const
                                  Event* ev) {
  compute(ev);
  return DiscriminatorRead::get();
}
```

Conclusion

The problem-specific points are now much fewer

- The object containing data (`Event`) in `fill` and `get` functions
- The list of variables in `EventDiscriminator.h`
- The registration of variables in `EventDiscriminator.cc`
- The creation of histograms in `EventDiscriminatorFill.cc`
- The variables calculation