Matrix Multiplication API Misused

Context

When the multiply operation is performed on two-dimensional matrixes, `np.matmul()` and `np.dot()` give the same result, which is a matrix.

Problem

In mathematics, the result of the dot product is expected to be a scalar rather than a vector. The `np.dot()` returns a new matrix for two-dimensional matrixes multiplication, which does not match with its mathematics semantics. Developers sometimes use `np.dot()` in scenarios where it is not supposed to, e.g., two-dimensional multiplication.

Solution

When the multiply operation is performed on two-dimensional matrixes, `np.matmul()` is preferred over `np.dot()` for its clear semantic.

Your dataset

file : EvalID_22/ppca.py

line : 113 ; 114

```
113:  res_x = x - np.dot(cvar, beta_x)
114:  res_y = y - np.dot(cvar, beta_y)
```

Should be a code smell

Dataframe Conversion API Misused

Context

In Pandas, `df.to_numpy()` and `df.values()` both can turn a `DataFrame` to a NumPy array.

Problem

As noted in a Stack Overflow post, `df.values()` has an inconsistency problem. With `.values()` it is unclear whether the returned value would be the actual array, some transformation of it, or one of the Pandas custom arrays. However, the `.values()` API has not been not deprecated yet. Although the library developers note it as a warning in the documentation, it does not log a warning or error when compiling the code if we use `.value()` .

Solution

When converting `DataFrame` to NumPy array, it is better to use `df.to_numpy()` than `df.values()` .

Your dataset

file : EvalID_14/LinearGaussianCPD.py

line : 133

```
beta_coef_matrix = np.matrix(coef_matrix.values, dtype="float")
```

Should be a code smell

Merge API Parameter Not Explicitly Set

Context

The `df.merge()` API merges two `DataFrame` s in Pandas.

Problem

Although using the default parameter can produce the same result, explicitly specify `on` and `how` produce better readability. The parameter `on` states which columns to join on, and the parameter `how` describes the join method (e.g., outer, inner). Also, the `validate` parameter will check whether the merge is of a specified type. If the developer assumes the merge keys are unique in both left and right datasets, but that is not the case, and he does not specify this parameter, the result might silently go wrong. The merge operation is usually computationally and memory expensive. It is preferable to do the merging process in one stroke for performance consideration.

Solution

Developer should explicitly specify the parameters for merge operation.

Your dataset

file : EvalID_54/kdd2012_track2_preprocess_data.py

line : 51

```
df
 .merge(df_title, on="title_id")
 .merge(df_query, on="query_id")
.merge(df_user, on="user_id", how="left")
 )
```

Report as a code smell in your dataset

but

NOT A CODE SMELL

Unecessary iterations

Context

Loops are typically time-consuming and verbose, while developers can usually use some vectorized solutions to replace the loops.

Problem

As stated in the Pandas documentation: “Iterating through pandas objects is generally slow. In many cases, iterating manually over the rows is not needed and can be avoided”. In `EffectiveTensorflow` github repository, it is also stated that the slicing operation with loops in TensorFlow is slow, and there is a substitute for better performance.

Solution

Machine learning applications are typically data-intensive, requiring operations on data sets rather than an individual value. Therefore, it is better to adopt a vectorized solution instead of iterating over data. In this way, the program runs faster and code complexity is reduced, resulting in more efficient and less error-prone code. Pandas’ built-in methods (e.g., join, groupby) are vectorized. It is therefore recommended to use Pandas built-in methods as an alternative to loops. In TensorFlow, using the `tf.reduce_sum()` API to perform reduction operation is much faster than combining slicing operation and loops.

Your dataset

file : EvalID_50/_m5.py

line : 135

for index, item in sales_train_validation.iterrows():

Should be a code smell

Your dataset

file : EvalID_79/protseq_analysis.py

line : 216

nuc_count[reads[j][i]] += 1

Report as a code smell in your dataset but

NOT A CODE SMELL

(not a pandas object)

NaN Equivalence Comparison Misused

Context

`NaN` equivalence comparison behaves differently from `None` equivalence comparison.

Problem

While `None == None` evaluates to `True`, `np.nan == np.nan` evaluates to `False` in NumPy. As Pandas treats `None` like `np.nan` for simplicity and performance reasons, a comparison of `DataFrame` elements with `np.nan` always returns `False`. If the developer is not aware of this, it may lead to unintentional bugs in the code.

Solution

Developers need to be careful when using the `NaN` comparison.

Your dataset

file : EvalID_77/DEP_xsolution.py

line : 103

```
if v_ is not np.nan and e_ is not np.nan:
```

Should be a code smell

Chain Indexing

Context

In Pandas, `df["one"]["two"]` and `df.loc[:,("one","two")]` give the same result. The `df["one"]["two"]` is called chain indexing.

Problem

Using chain indexing may cause performance issues as well as prone-to-bug code. For example, when using `df["one"]["two"]` , Pandas sees this operation as two events: call `df["one"]` first and call `["two"]` based on the result the previous operation gets. On the contrary, `df.loc[:,("one","two")]` only perform a single call. In this way, the second approach can be significantly faster than the first one. Furthermore, assigning to the product of chain indexing has inherently unpredictable results. Since Pandas makes no guarantees on whether `df["one"]` will return a view or a copy, the assignment may fail.

Solution

Developers using Pandas should avoid using chain indexing.

Your dataset

file : EvalID_6/report.py

line : 115

not a dataframe !

metrics["precision"]["accuracy"] = accuracy

Report as a code smell in your dataset

but

NOT A CODE SMELL

Columns and DataType Not Explicitly Set

Context

In Pandas, all columns are selected by default when a `DataFrame` is imported from a file or other sources. The data type for each column is defined based on the default `dtype` conversion.

Problem

If the columns are not selected explicitly, it is not easy for developers to know what to expect in the downstream data schema. If the datatype is not set explicitly, it may silently continue the next step even though the input is unexpected, which may cause errors later. The same applies to other data importing scenerios.

Solution

It is recommended to set the columns and `DataType` explicitly in data processing.

Your dataset

file : EvalID_50/_m5.py

line : 50

Should be a code smell
(no “dtype”)

```
sell_prices = pd.read_csv(sell_prices_path, index_col=["item_id", "store_id"])
```

TensorArray Not Used

Developers may need to change the value of the array in the loops in TensorFlow.

Problem

If the developer initializes an array using `tf.constant()` and tries to assign a new value to it in the loop to keep it growing, the code will run into an error. The developer can fix this error by the low-level `tf.while\_loop()` API. However, it is inefficient coding in this way. A lot of intermediate tensors are built in this process.

Solution

Using `tf.TensorArray()` for growing array in the loop is a better solution for this kind of problem in TensorFlow 2.

Your dataset

file : EvalID_71/transforms.py

line : 53

```
47:    :param samples: Sample arrays
48:    :param labels: Label arrays
49:    :param mean: Mean (unused)
50:    :param stdev: Std (unused)
51:    :return: Padded samples and labels
52:    """
53:    paddings = tf.constant([[0, 0], [0, 0], [0, 5], [0, 0]])
54:    samples = tf.pad(samples, paddings, "CONSTANT")
55:    if labels is None:
56:      return samples
57:    labels = tf.pad(labels, paddings, "CONSTANT")
58:    return samples, labels
```

Report as a code smell in your dataset

but

NOT A CODE SMELL

because

don't respect the condition of the smell (loop)

Pytorch Call Method Misused

Context

Both `self.net()` and `self.net.forward()` can be used to forward the input into the network in PyTorch.

Problem

In PyTorch, `self.net()` and `self.net.forward()` are not identical. The `self.net()` also deals with all the register hooks, which would not be considered when calling the plain `.forward()` .

Solution

It is recommended to use `self.net()` rather than `self.net.forward()` .

Your dataset

file : EvalID_66/ac.py

line : 160

Should be a code smell

```
mean, log_std = self.forward(state)
```

Pytorch Call Method Misused

Context

Both `self.net()` and `self.net.forward()` can be used to forward the input into the network in PyTorch.

Problem

In PyTorch, `self.net()` and `self.net.forward()` are not identical. The `self.net()` also deals with all the register hooks, which would not be considered when calling the plain `.forward()` .

Solution

It is recommended to use `self.net()` rather than `self.net.forward()` .

Your dataset

file : EvalID_66/ac.py

line : 160

Should be a code smell

```
mean, log_std = self.forward(state)
```

What's the issue?

In PyTorch, neural network classes usually inherit from *torch.nn.Module*.

For those, you're supposed to call the module like a function:

```
out = model(x)
```

Not by calling its forward method directly:

```
out = model.forward(x)
```

Why? Because *nn.Module.__call__* wraps ***forward*** and does important extra work (e.g., runs registered forward hooks, cooperates with wrappers like DataParallel/DDP, tracing/compilers, some profilers/instrumentation).

If you **skip** *__call__* by **invoking** *forward* yourself, all that extra behavior is silently bypassed.

In the snippet:

```
probs = self.forward(state)
probs = self(state)
mean, log_std = self.forward(state)
mean, log_std = self(state)
```

The minimal fix

Replace with calls to the module itself

Gradients Not Cleared before Backward Propagation

Context

In PyTorch, `optimizer.zero_grad()` clears the old gradients from last step, `loss_fn.backward()` does the back propagation, and `optimizer.step()` performs weight update using the gradients.

Problem

If `optimizer.zero_grad()` is not used before `loss_fn.backward()`, the gradients will be accumulated from all `loss_fn.backward()` calls and it will lead to the gradient explosion, which fails the training.

Solution

Developers should use `optimizer.zero_grad()`, `loss_fn.backward()`, `optimizer.step()` together in order and should not forget to use `optimizer.zero_grad()` before `loss_fn.backward()`.

Your dataset

file : EvalID_30/train.py_

line : 160

Should be a code smell
(not the right order)

```
159:    loss = output["loss"]
160:    loss.backward()
161:    ds_loss[0] += loss.item()
162:    ds_loss[1] += len(batch["input_ids"])
163:    optimizer.zero_grad()
```