Adding a new model

This folder contains templates to generate new models that fit the current API and pass all tests. It generates models in both PyTorch, TensorFlow, and Flax and completes the `__init__.py` and auto-modeling files, and creates the documentation. Their use is described in the next section.

There is also a CLI tool to generate a new model like an existing one called `transformers-cli add-new-model-like`. Jump to the Add new model like section to learn how to use it.

Cookiecutter Templates

Using the `cookiecutter` utility requires to have all the `dev` dependencies installed. Let's first clone the repository and install it in our environment:

```
git clone https://github.com/huggingface/transformers
cd transformers
pip install -e ".[dev]"
```

Once the installation is done, you can use the CLI command `add-new-model` to generate your models:

```
transformers-cli add-new-model
```

This should launch the `cookiecutter` package which should prompt you to fill in the configuration.

The `modelname` should be cased according to the plain text casing, i.e., BERT, RoBERTa, DeBERTa.

```
modelname [<ModelNAME>]:
uppercase_modelname [<MODEL_NAME>]:
lowercase_modelname [<model_name>]:
camelcase_modelname [<ModelName>]:
```

Fill in the `authors` with your team members:

```
authors [The HuggingFace Team]:
```

The checkpoint identifier is the checkpoint that will be used in the examples across the files. Put the name you wish, as it will appear on the modelhub. Do not forget to include the organisation.

```
checkpoint_identifier [organisation/<model_name>-base-cased]:
```

The tokenizer should either be based on BERT if it behaves exactly like the BERT tokenizer, or a standalone otherwise.

```
Select tokenizer_type:
1 - Based on BERT
2 - Standalone
Choose from 1, 2 [1]:
```

Once the command has finished, you should have a total of 7 new files spread across the repository:

```
docs/source/model_doc/<model_name>.mdx
src/transformers/models/<model_name>/configuration_<model_name>.py
src/transformers/models/<model_name>/modeling_<model_name>.py
src/transformers/models/<model_name>/modeling_tf_<model_name>.py
src/transformers/models/<model_name>/tokenization_<model_name>.py
tests/test_modeling_<model_name>.py
tests/test_modeling_tf_<model_name>.py
```

You can run the tests to ensure that they all pass:

```
python -m pytest ./tests/test_*<model_name>*.py
```

Feel free to modify each file to mimic the behavior of your model.

⚠ You should be careful about the classes preceded by the following line:

```
# Copied from transformers.[...]
```

This line ensures that the copy does not diverge from the source. If it *should* diverge, because the implementation is different, this line needs to be deleted. If you don't delete this line and run `make fix-copies`, your changes will be overwritten.

Once you have edited the files to fit your architecture, simply re-run the tests (and edit them if a change is needed!) afterwards to make sure everything works as expected.

Once the files are generated and you are happy with your changes, here's a checklist to ensure that your contribution will be merged quickly:

- You should run the `make fixup` utility to fix the style of the files and to ensure the code quality meets the library's standards.
- You should complete the documentation file (`docs/source/model_doc/<model_name>.rst`) so that your model may be usable.

Add new model like command

Using the `transformers-cli add-new-model-like` command requires to have all the `dev` dependencies installed. Let's first clone the repository and install it in our environment:

```
git clone https://github.com/huggingface/transformers
cd transformers
pip install -e ".[dev]"
```

Once the installation is done, you can use the CLI command `add-new-model-like` to generate your models:

```
transformers-cli add-new-model-like
```

This will start a small questionnaire you have to fill.

```
What identifier would you like to use for the model type of this model?
```

You will have to input the model type of the model you want to clone. The model type can be found in several places:

- inside the configuration of any checkpoint of that model
- the name of the documentation page of that model

For instance the doc page of `BigBirdPegasus` is `https://huggingface.co/docs/transformers/model_doc/bigbird_pegasus` so its model type is `"bigbird_pegasus"`.

If you make a typo, the command will suggest you the closest model types it can find.

Once this is done, the questionnaire will ask you for the new model name and its various casings:

```
What is the name for your new model?
What identifier would you like to use for the model type of this model?
What name would you like to use for the module of this model?
What prefix (camel-cased) would you like to use for the model classes of this model?
What prefix (upper-cased) would you like to use for the constants relative to this
model?
```

From your answer to the first question, defaults will be determined for all others. The first name should be written as you want your model be named in the doc, with no special casing (like RoBERTa) and from there, you can either stick with the defaults or change the cased versions.

Next will be the name of the config class to use for this model:

```
What will be the name of the config class for this model?
```

Then, you will be asked for a checkpoint identifier:

```
Please give a checkpoint identifier (on the model Hub) for this new model.
```

This is the checkpoint that will be used in the examples across the files and the integration tests. Put the name you wish, as it will appear on the Model Hub. Do not forget to include the organisation.

Then you will have to say whether your model re-uses the same processing classes as the model you're cloning:

```
Will your new model use the same processing class as Xxx
(XxxTokenizer/XxxFeatureExtractor)
```

Answer yes if you have no intentions to make any change to the class used for preprocessing. It can use different files (for instance you can reuse the `BertTokenizer` with a new vocab file).

If you answer no, you will have to give the name of the classes for the new tokenizer/feature extractor/processor (depending on the model you're cloning).

Next the questionnaire will ask

```
Should we add # Copied from statements when creating the new modeling file?
```

This is the intenal mechanism used in the library to make sure code copied from various modeling files stay consistent. If you plan to completely rewrite the modeling file, you should answer no, whereas if you just want to tweak one part of the model, you should answer yes.

Lastly, the questionnaire will inquire about frameworks:

```
Should we add a version of your new model in all the frameworks implemented by Old
Model (xxx)?
```

If you answer yes, the new model will have files for all the frameworks implemented by the model you're cloning. Otherwise, you will get a new question to select the frameworks you want.

Once the command has finished, you will see a new subfolder in the `src/transformers/models/` folder, with the necessary files (configuration and modeling files for all frameworks requested, and maybe the processing files, depending on your choices).

You will also see a doc file and tests for your new models. First you should run

```
make style
maxke fix-copies
```

and then you can start tweaking your model. You should:

- fill the doc file at `docs/source/model_doc/model_name.mdx`
- tweak the configuration and modeling files to your need

Once you're done, you can run the tests to ensure that they all pass:

```
python -m pytest ./tests/test_*<model_name>*.py
```

⚠ You should be careful about the classes preceded by the following line:

```
# Copied from transformers.[...]
```

This line ensures that the copy does not diverge from the source. If it *should* diverge, because the implementation is different, this line needs to be deleted. If you don't delete this line and run `make fix-copies`, your changes will be overwritten.

Once you have edited the files to fit your architecture, simply re-run the tests (and edit them if a change is needed!) afterwards to make sure everything works as expected.

Once the files are generated and you are happy with your changes, here's a checklist to ensure that your contribution will be merged quickly:

- You should run the `make fixup` utility to fix the style of the files and to ensure the code quality meets the library's standards.
- You should add your model to the main README then run `make fix-copies`.