AInterface

Udacity Machine Learning Nanodegree Capstone Project

Extract

This report will present a solution to two different problems. First is the Numerai data competition which provides a data set and requires the best possible predictions. The second problem is need for a uniform interface for any and all machine learning algorithms. This reports provides a structure and data model to provide this uniform interface and implements multiple models. This uniform structure is then used in the Numerai data competition to create predictions with a correlation coefficient of 5.8459%, compared to a default linear learner model that results in a correlation coefficient of 0.9468%.

Definition of the problem

The problem to face is twofold, a practical problem to produce the best predictions for the Numerai competition, and a more high-level problem to have a consistent interface for any machine learning algorithm.

Practical – The Numerai competition

Numerai is a crowd sourced, hedge fund that has a competition to predict stock markets with the help of the crowd. Numerai provides an anonymised data set of 311 feature column and 1 output columns, and around half a million data points. Every week they provide a new set of tournament data which the competitors have to predict.

The goal will be to create a program that can:

1. Down the latest tournament data automatically from Numerai
2. Train a model based on the training data
3. Use the trained model to predict the tournament data
4. Upload the predictions to Numerai automatically to enter the competition

High level – A consistent machine learning interface

During the other courses of the Nanodegree, we used several different algorithms and frameworks. All these algorithms have different methods, require different input data format and provide different output formats. As I wasn't working on the project all the time, I often forgot which format the algorithm was expecting, and how to exactly provide it.

The goal of this second sub-problem is to create a uniform interface for many different algorithms. The user should be able to load the data, and train any model defined in the program in exactly the same way. Internally, the model will then format the data as necessary, absolving the user of any fiddling around.

For lack of a better name, I have called this part AInterface.

Analysis of the problem

We will first analyse the requirements of the AInterface part and then continue to the Numerai data data set and competition.

AInterface

To provide a uniform interface, we first need to decomposition what a machine algorithm does.

General structure

Any machine learning process contains at least these three parts (Figure 1):

- Loading data
- Training the model
- Predicting new data points

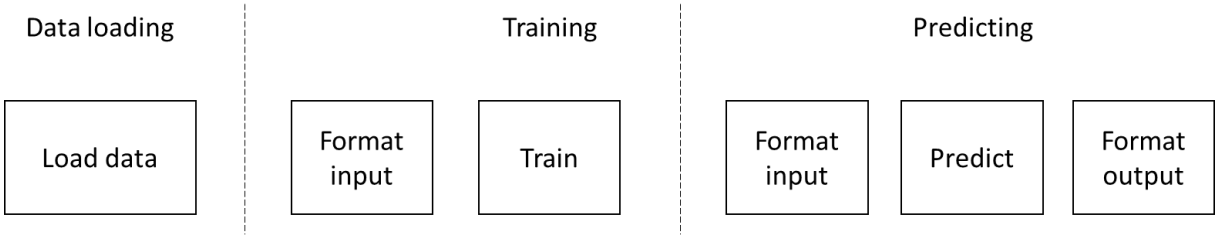

Figure 1: Machine learning general structure

The data loading step is quite straight forward. In our case, the data is in a CSV file, which can be read in the program. Other possibilities are databases (SQL, NoSQL), unformatted data, text data etc. This data can also be preprocessed, but this will be out of scope of this project. However, this could be a later feature of the solution. The data loader could also help with train, validation, test splitting, and selection of features and outputs.

The second step is the training of the model. Abstracting how the model actually works, this can be split into two steps:

- Formatting the input
- Training the model with the provided input

In general, training doesn't provide a direct output, except for a trained model which can be used later. This storing of the so-called artefacts are considered out of scope of this project, as this is done very well by current solutions (often tar or gz files). When you already have a trained model, this step can be skipped to go straight to prediction by loading these artefacts.

The last step is predicting outputs. Again, the model requires formatted input. The model uses its artefacts to provide predictions and provides a specific type of output. Depending on the rest of your logic, you might need to reformat this output to a format that you can use.

The goal of the solution is to create a uniform structure for these three methods. The data loader should load the data in a way that can be used over the rest of the algorithm. The new model structure should accept this format and transform it internally to whatever format the algorithm

requires it to be. For predictions, it will also transform the output to the general format used in the rest of the program.

Numerai

Data

Each data set has 310 features, divided in 6 types (*intelligence*, *charisma*, *strength*, *dexterity*, *constitution*, *wisdom*), 1 output variable (*target_kazutsugi*), an id, an time variable (era) and a data type (train, validation, test, live).

The training data set is 501808 rows with only training data, the tournament data is 385290 rows, divided on validation, test and live data.

Each feature is normalised and one of 0.00, 0.25, 0.50, 0.75 and 1.00. The target has the same values. The era is between 1-120 for training data, 121-132 for validation data, 133-196 for test data and eraX for live data.

Numerai uses the correlation coefficient to rate its submissions (`np.corrcoef`).

Models

Because all numerical data is a multiple of 0.25, including the output features, and because these multiples are also meant as numerical data, both classification models as regression models can be used.

Automation

Numerai provides a python packages, Numerapi, that helps automate the competition. It can download and upload the data for you as long as you provide the correct input formats. We will use this interface to automate the competition.

Implementation a solution

The goals of the total solution is an automated program. This is why there are only normal python files and programs. There are also two example python files, and a short Jupyter Notebook.

Introduction

Coding styles

While Python is a dynamically typed language, recent version include optional styling. And for the AInterface part of the problem, we want to have a uniform interface for our solution. This is why I chose to use the typing library of Python to help implement typing. MyPy is used to check for inconsistencies, but not 100% followed.

Secondly, I chose to use Black for a coding style, as it helps by automatically formatting the code.

Logging and errors

There is a basic implementation of logging in the program, although it could be improved. There is a bare minimum of error handling.

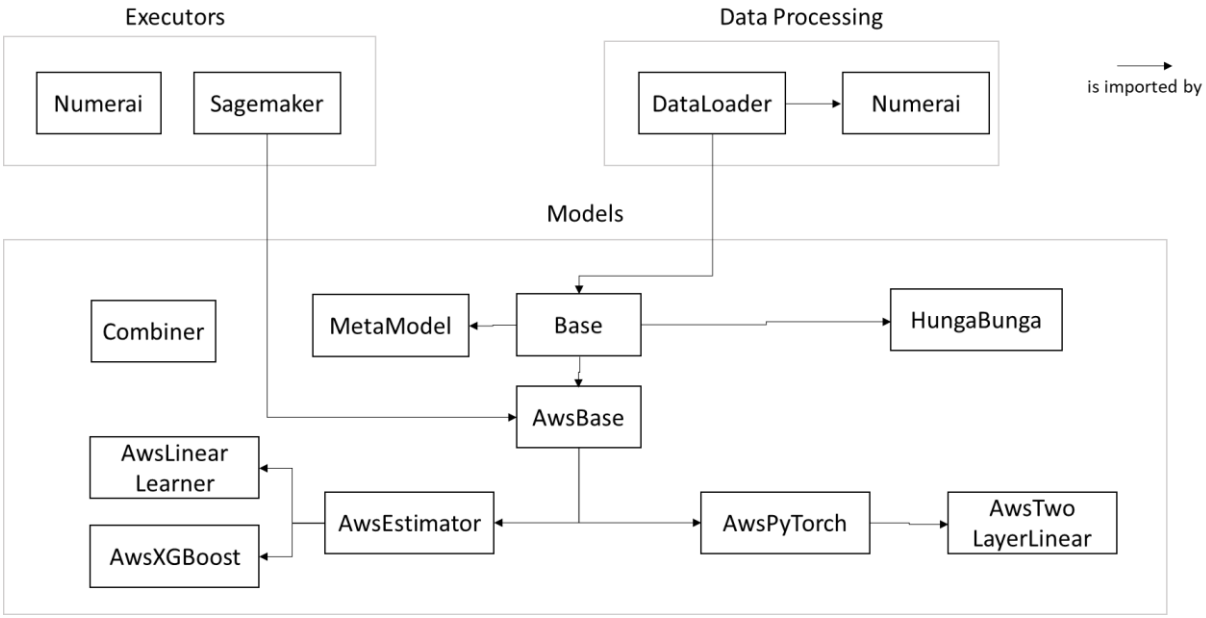

Figure 2: Data model

Data Model

Figure 2 provides the data model of the implementation. Each domain corresponds to its own folder. Each class roughly corresponds to its own python file, with the exception of AwsEstimator, which is in the *aws_base.py* file and *AwsTwoLayerLinear* which is in the *pytorch/aws_models.py* file. That specific implementation does provide the *two_layer_linear.py* as its training file.

Data processing

DataLoader

DataLoader is the base class for processing and providing the data. It loads the data either from a csv, or from a DataFrame directly. Data is loaded lazily, so is not used until it is referred.

It provides train, validation and test data split out of the box. It also internally stores which columns are used as features and which columns as output.

It has a way to split itself into multiple batches, for when you would to split the data into different data loader to train individual models.

Lastly, models can add different formats to an internal cache. For example, if a model formats the DataFrame to a local CSV, it can add it to the cache. Another model will later be able to retrieve the data from the cache, avoiding the need to save it to CSV twice.

The DataLoader itself is an Abstract class and can nor should not be initialised directly. A subclass should at least implement *output_column* and *feature_columns*.

Numerai (DataLoader)

The Numerai implementation of the DataLoader sets the feature columns to those of the Numerai data set. It also provides a scoring method and a way to format the predictions for later use.

Executors

Sagemaker

The Sagemaker executor provides the interface with Amazon. It creates a session, and stores the default buckets, default prefixes, and default parameters for any model, transformer and predictors. This way, the implemented Aws classes don't need to implement these individually, although they can overwrite them.

Next to storing defaults, it also provides a method to upload and download data to and from S3. It expects a local file for uploading, and stores to a local file for downloading. In itself it's a dumb component and does no checks on formatting.

Numerai (Executor)

The Numerai executor provides an interface to the Numerapi library. It can download and upload the predictions, as well as format a DataFrame to the correct CSV format that Numerai expects.

Models

BaseModel

The BaseModel provides the basic interface that any model should extend. This includes a *train*, *tune*, and *predict*/*execute_prediction* function.

The *predict* and *execute_predcition* functions are split. The *predict* does some general administration, extracting the correct data, while the *execute_prediction* only has the task of predicting an input DataFrame and outputting the prediction as a DataFrame.

AwsBase

The AwsBase class is again an abstract class, but provides basic functionalities to upload and download data from S3.

AwsEstimator

The AwsEstimator class provides a first actual implementation. The internal model of this class is the Estimator class from Amazon. This Estimator loads a model from AWS, for example XG Boost or a LinearLearner.

The Amazon Estimator class has a specific flow. It first loads a Model which can be trained. Next, the model can either be used for batch processing. Or it can be deployed online with a predictor. Both these methods have a consistent implementation from Amazon and this can be combined.

The execution follows the following process

- Train
    - Load the model with either input parameters, of defaults
    - Format the data and upload to S3 (and add to caches)
    - Call the training function with hyperparameters
- Prediction
    - Optional: load an existing model
    - Format the data and upload to S3
    - Create a transformer
    - Do the predictions
    - Load the data from S3
    - Format the csv file and return a DataFrame

The choice for a Transformer was done as for Numerai, one predictions per week will be done with a large number of data points. This is better than a Predictor, which is more useful for constant, small sized predictions.

A *tune* method is also provided, to create an optimized model.

AwsXGBoost

AwsXGBoost, LinearAwsXGBoost, BinaryAwsXGBoost and MulticlassAwsXGBoost are implementations of the AwsEstimator specifically for the XG boost algorithm. It sets the image name, local names and default hyperparameters for the specific models.

AwsLinearLearner

AwsLinearLearner, LinearAwsLinearLearner, BinaryAwsLinearLearner and MulticlassAwLinearLearner are implementations of the AwsEstimator specifically for the linear regression algorithm. It sets the image name, local names and default hyperparameters for the specific models.

AwsPyTorch

The Pytorch implementation on Amazon works quite differently from the Estimator model.

- The model refers not to an image but to a training/prediction file which contains all the logic. The implementation of this file is discussed laster.
- The training data is expected with an s3 location.
- There is not batch transformation, only a prediction endpoint.
    - The data must be provided with direct Numpy arrays.
    - The input data can be maximum 5Mb and thus needs to be split up and reassembled.

Regarding the train and predict file. As a lot of the code is similar, these have been merged into one file. Amazon places specific restrictions on the file as well.

- It must be standalone, meaning it cannot have local imports.
- For training, the file is called directly, thus *if __name == '__main__'* can be checked
- For prediction, the *predict_fn* is called, as well as *input_fn* and *output_fn*.

AwsTwoLayerLinearNeuralNetwork

The neural network is a simple two layered linear neural network. Input, output and hidden dimensions are its hyperparameters. This class is the local model class that has the correct interface for locally.

The implementation of the actual PyTorch (models/pytorch/two_layer_linear) is heavily based on the examples provided in one of the previous projects. The main change has been the model itself at the top.

The idea is that all local models would reside in models/pytorch/aws_models.py, while the implementations would each get a different file in the same folder.

While this is a simplified model, it should be easy to implement new models, by creating a new (local) AwsPyTorch class, copying the implementation file and modifying the model and the argument parsing.

Hunga Bunga

Hunga Bunga is an interface on all the SKLearn models, both for classification and regression. The local HungaBunga models integrate these models locally so they can also be used.

MetaModel

A MetaModel combine different models to create an event better model. It requires different models as input, and a combination method which it will use. Implementation of each of the methods is quite straight forward.

The train method, trains each of its models individually. During training it will also use the test data to determine its weights, by predicting the data and then combining it.

Predicting work similarily, by predicting with each of its models and then combining it with the weights it has set internally.

Combiner

The Combiner function works in the MetaModel to calculate the weights of each model in the MetaModel. The implementation of the NaiveCombiner is a grid search of the models to calculate the best score. It requires a combination function which takes two DataFrames and outputs a score. Currently only maximising score is supported.

Bringing it all together

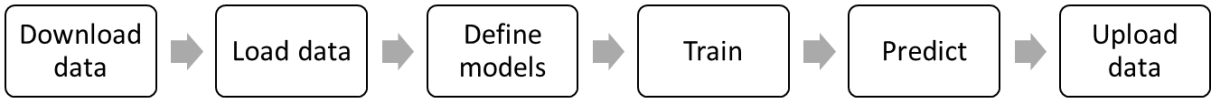

Figure 3: End to end flow

For the Numerai competition this will be the flow of the program (Figure 3). This flow is defined in examples/example_training.py

- Download data
  - Download data from the Numerai competition
- Load data
  - In the NumeraiDataLoader
- Define models
  - Define multiple individual models
  - Define a MetaModel with those models and a Combiner
- Train data
- Predict
  - From the tournament data
- Upload data
  - To the competition

The goals is, once the models have been defined, implementation is very easy. Additionally, switching around a DataLoader or a Model should be straightforward as long as it follows the defined interfaces.

Collection of results

The collection of results will focus on the Numerai competition, and less on AInterface.

The collection of results is also done in the *notebook* Jupyter file in the repository.

Benchmark

For a benchmark we will use a default Linear Learner algorithm, without any tuning and using the default hyperparameters. As the AInterface part of the project was more focused on ease of use, this is very easy to implement:

```
 # Loading data and setting up sagemaker

numerai = Numerai()
data_location = numerai.download_latest_data("data/temp")
sagemaker = Sagemaker()
data_loader = NumeraiDataLoader(
    local_data_location=os.path.join(data_location,
"numerai_training_data.csv")
)

# Do Machine Learning
linear_learner = LinearAwsLinearLearner(data=data_loader,
aws_executor=sagemaker)
linear_learner.train()
predictions = linear_learner.predict(all_data=False)
score = data_loader.score(predictions, all_data=False)
```

The score (correlation coefficient) for this default model is 0.9468%

MetaModel

Using the MetaModel with each of the Amazon models that are implemented, the code barely looks any different. More models are defined, as well as a MetaModel and a Combiner:

```
# Loading data and setting up sagemaker
numerai = Numerai()
data_location = numerai.download_latest_data("data/temp")
sagemaker = Sagemaker()
data_loader = NumeraiDataLoader(
    local_data_location=os.path.join(data_location,
"numerai_training_data.csv")
)

# Loading models and Predictors
xgboost = LinearAwsXGBooost(data=data_loader, aws_executor=sagemaker)
linear_learner = LinearAwsLinearLearner(data=data_loader,
aws_executor=sagemaker)
tl_nn = AwsTwoLayerLinearNeuralNetwork(data=data_loader,
aws_executor=sagemaker)

# Creating the meta model to combine them
combiner = NaiveCombiner(data_loader.execute_scoring, 10)
meta_model = MetaModel(
    data=data_loader, models=[xgboost, linear_learner, tl_nn],
combiner=combiner
)

# Train/tune and set the weights
meta_model.tune()
predictions = linear_learner.predict(all_data=False)
score = data_loader.score(predictions, all_data=False)
```

The score for this more complicated function is 5.8459%.

Comparing this to previous results of the Numerai competition, where the results are around 2.2%. A note must be made that the competition itself relies on another data source and that result will probably be worse than with the test data. Because of the nature of the competition (a hedge fund) the production data will probably be newer data and less correlated with the data in the training set which will probably be of the same time period. This is also implied in the era column of the data.

Loading trained models

To load already trained models, the load functions are added and the train function is removed. This results in the following code:

```
# Load existing models
xgboost = LinearAwsXGBooost(data=data_loader, aws_executor=sagemaker)
xgboost.load_model("xgboost-190911-2045-001-9ea55a3c")
linear_learner = LinearAwsLinearLearner(data=data_loader,
aws_executor=sagemaker)
linear_learner.load_model("linear-learner-190911-2051-010-c1988378")
tl_nn = AwsTwoLayerLinearNeuralNetwork(data=data_loader,
aws_executor=sagemaker)
tl_nn.load_estimator("sagemaker-pytorch-2019-09-11-19-47-58-170")
```

Automated Numerai flow

To include the Numerai flow, only this simple addition needs to be made, which uses the production data, predicts it and uploads the formatted data to Numerai.

```
# Predict Numerai production data
production_data = NumeraiDataLoader(
    local_data_location=os.path.join(data_location,
"numerai_tournament_data.csv")
)
predictions = meta_model.predict(data_loader=production_data,
all_data=True)

# Format and upload
predictions = production_data.format_predictions(predictions,
all_data=True)
numerai.upload_predictions(predictions,
local_folder="data/temp/predictions")
```

As with the AInterface, ease of use was the highest priority.

Conclusion

This project was focused on two major parts, being the Numerai competition, and AInterface, a uniform interface for any machine learning model.

The result of the Numerai competition involved a combination of three optimised models, a Linear Learner, XGBoost and a PyTorch neural network. All algorithms are run on Aws Sagemaker. A basic Linear Learner got a 0.9468% result, while the combined model had 5.8549% result.

The goal of the AInterface is to provide a uniform interface for any algorithm such that models can be mixed and matched, data can be swapped around and extensions can be easily made. For this a structure of DataLoaders, Models, Executors and Combiners. The in- and output of these parts were set, and several implementation were made: the LinearLearner and XGBoost estimators, and a

PyTorch model, all three on Amazon and a Numerai DataLoader. These provide a simple to use interface for which the data scientist, analyst or machine learning engineer can focus on making the best algorithm, instead of integrating different services.

Next steps

Additional features

- More models
  - Implementation of Classifications
  - TensorFlow
  - Keras
- More combiners
- Different platforms
  - TensorFlow, Keras and PyTorch locally
  - Google Cloud
  - Microsoft Azure
- Data preprocessing
  - PCA
- States
  - State creation
  - State loading
  - Default configuration
  - State validation

Improvement points

- Refactoring of the PyTorch models (the actual implementation files)
- Refactoring of hyperparameters
- Parallel training or predictions
  - For non-local models
  - For local models
- Error handling
- Logging

Additional information

Used libraries

- Numpy, Pandas, Sklearn: General data science libraries
- Boto3, Sagemaker: Connection with AWS
- Numerapi: Connection with Numerai
- Hunga Bunga, tabulate: implementation layer on top of SKLearn, usefull for trying a lot of models
- TorchVision: for neural networks

Development packages

- Black: automatic formatter
- MyPy: helps with typing

- Jupyter: for data exploration

Numerai background

(Same as proposal)

Numerai is an investment company that outsources (a part of) its research to the community. They host a competition and provide rewards for the best predictions of a data set that they provide.

They provide a clean dataset of 1 time variable (era), 310 normalised features and 1 output variable. The training dataset is around 500.000 lines, and they also provide validation, testing and live data. The output variable is required as a fraction, as thus makes it a regression problem.

The goal is to predict the live and test set, which only Numerai will validate. If you join the competition your predictions will be compared to other community members.

You can either just compete or you can also stake some numer, a cryptocurrency related to Etherium. If you stake some numer, you can earn more numer if your predictions are better than average. The exact details for how and how much you earn can be found here.

Links

Project proposal review: https://review.udacity.com/#!/reviews/1933269

Numerai: https://numer.ai/

Numerapi: https://github.com/uuazed/numerapi

Hunga Bunga: https://github.com/ypeleg/HungaBunga

MyPy: https://mypy.readthedocs.io/

Typing: https://docs.python.org/3/library/typing.html

Black: https://pypi.org/project/black/