Eps

Machine learning for Ruby

- Build predictive models quickly and easily
- Serve models built in Ruby, Python, R, and more

Check out this post for more info on machine learning with Rails

build passing

Installation

Add this line to your application’s Gemfile:

```ruby
gem "eps"
```

On Mac, also install OpenMP:

```sh
brew install libomp
```

Getting Started

Create a model

```ruby
data = [
  {bedrooms: 1, bathrooms: 1, price: 100000},
  {bedrooms: 2, bathrooms: 1, price: 125000},
  {bedrooms: 2, bathrooms: 2, price: 135000},
  {bedrooms: 3, bathrooms: 2, price: 162000}
]
model = Eps::Model.new(data, target: :price)
puts model.summary
```

Make a prediction

```ruby
model.predict(bedrooms: 2, bathrooms: 1)
```

Store the model

```ruby
File.write("model.pmml", model.to_pmml)
```

Load the model

```ruby
pmml = File.read("model.pmml")
model = Eps::Model.load_pmml(pmml)
```

A few notes:

- The target can be numeric (regression) or categorical (classification)
- Pass an array of hashes to `predict` to make multiple predictions at once
- Models are stored in PMML, a standard for model storage

Building Models

Goal

Often, the goal of building a model is to make good predictions on future data. To help achieve this, Eps splits the data into training and validation sets if you have 30+ data points. It uses the training set to build the model and the validation set to evaluate the performance.

If your data has a time associated with it, it's highly recommended to use that field for the split.

```
Eps::Model.new(data, target: :price, split: :listed_at)
```

Otherwise, the split is random. There are a number of other options as well.

Performance is reported in the summary.

- For regression, it reports validation RMSE (root mean squared error) - lower is better
- For classification, it reports validation accuracy - higher is better

Typically, the best way to improve performance is feature engineering.

Feature Engineering

Features are extremely important for model performance. Features can be:

1. numeric
2. categorical
3. text

Numeric For numeric features, use any numeric type.

```
{bedrooms: 4, bathrooms: 2.5}
```

Categorical For categorical features, use strings or booleans.

```
{state: "CA", basement: true}
```

Convert any ids to strings so they're treated as categorical features.

```
{city_id: city_id.to_s}
```

For dates, create features like day of week and month.

```
{weekday: sold_on.strftime("%a"), month: sold_on.strftime("%b")}
```

For times, create features like day of week and hour of day.

```
{weekday: listed_at.strftime("%a"), hour: listed_at.hour.to_s}
```

Text For text features, use strings with multiple words.

```
{description: "a beautiful house on top of a hill"}
```

This creates features based on word count.

You can specify text features explicitly with:

```
Eps::Model.new(data, target: :price, text_features: [:description])
```

You can set advanced options with:

```
text_features: {
  description: {
    min_occurences: 5,          # min times a word must appear to be
        included in the model
    max_features: 1000,         # max number of words to include in the
         model
    min_length: 1,              # min length of words to be included
    case_sensitive: true,       # how to treat words with different
        case
    tokenizer: /\s+/,           # how to tokenize the text, defaults to
         whitespace
    stop_words: ["and", "the"]  # words to exclude from the model
  }
}
```

Full Example

We recommend putting all the model code in a single file. This makes it easy to rebuild the model as needed.

In Rails, we recommend creating a `app/ml_models` directory. Be sure to restart Spring after creating the directory so files are autoloaded.

```
bin/spring stop
```

Here's what a complete model in `app/ml_models/price_model.rb` may look like:

```ruby
class PriceModel < Eps::Base
  def build
    houses = House.all

    # train
    data = houses.map { |v| features(v) }
    model = Eps::Model.new(data, target: :price, split: :listed_at)
    puts model.summary

    # save to file
    File.write(model_file, model.to_pmml)

    # ensure reloads from file
    @model = nil
  end

  def predict(house)
    model.predict(features(house))
  end

  private

  def features(house)
    {
      bedrooms: house.bedrooms,
      city_id: house.city_id.to_s,
      month: house.listed_at.strftime("%b"),
      listed_at: house.listed_at,
      price: house.price
    }
  end

  def model
    @model ||= Eps::Model.load_pmml(File.read(model_file))
  end

  def model_file
    File.join(__dir__, "price_model.pmml")
  end
end
```

Build the model with:

```ruby
PriceModel.build
```

This saves the model to `price_model.pmml`. Check this into source control or use a tool like Trove to store it.

Predict with:

```
PriceModel.predict(house)
```

Monitoring

We recommend monitoring how well your models perform over time. To do this, save your predictions to the database. Then, compare them with:

```
actual = houses.map(&:price)
predicted = houses.map(&:predicted_price)
Eps.metrics(actual, predicted)
```

For RMSE and MAE, alert if they rise above a certain threshold. For ME, alert if it moves too far away from 0. For accuracy, alert if it drops below a certain threshold.

Other Languages

Eps makes it easy to serve models from other languages. You can build models in Python, R, and others and serve them in Ruby without having to worry about how to deploy or run another language.

Eps can serve LightGBM, linear regression, and naive Bayes models. Check out ONNX Runtime and Scoruby to serve other models.

Python

To create a model in Python, install the sklearn2pmml package

```
pip install sklearn2pmml
```

And check out the examples:

- LightGBM Regression
- LightGBM Classification
- Linear Regression
- Naive Bayes

R

To create a model in R, install the pmml package

```r
install.packages("pmml")
```

And check out the examples:

- Linear Regression
- Naive Bayes

Verifying

It’s important for features to be implemented consistently when serving models created in other languages. We highly recommend verifying this programmatically. Create a CSV file with ids and predictions from the original model.

house_id	prediction
1	145000
2	123000
3	250000

Once the model is implemented in Ruby, confirm the predictions match.

```ruby
model = Eps::Model.load_pmml("model.pmml")

# preload houses to prevent n+1
houses = House.all.index_by(&:id)

CSV.foreach("predictions.csv", headers: true, converters: :numeric) do
    |row|
  house = houses[row["house_id"]]
  expected = row["prediction"]

  actual = model.predict(bedrooms: house.bedrooms, bathrooms: house.
      bathrooms)

  success = actual.is_a?(String) ? actual == expected : (actual -
      expected).abs < 0.001
  raise "Bad prediction for house #{house.id} (exp: #{expected}, act:
      #{actual})" unless success

  putc "✓"
```

```ruby
end
```

Data

A number of data formats are supported. You can pass the target variable separately.

```ruby
x = [{x: 1}, {x: 2}, {x: 3}]
y = [1, 2, 3]
Eps::Model.new(x, y)
```

Data can be an array of arrays

```ruby
x = [[1, 2], [2, 0], [3, 1]]
y = [1, 2, 3]
Eps::Model.new(x, y)
```

Or Numo arrays

```ruby
x = Numo::NArray.cast([[1, 2], [2, 0], [3, 1]])
y = Numo::NArray.cast([1, 2, 3])
Eps::Model.new(x, y)
```

Or a Rover data frame

```ruby
df = Rover.read_csv("houses.csv")
Eps::Model.new(df, target: "price")
```

Or a Daru data frame

```ruby
df = Daru::DataFrame.from_csv("houses.csv")
Eps::Model.new(df, target: "price")
```

When reading CSV files directly, be sure to convert numeric fields. The `table` method does this automatically.

```ruby
CSV.table("data.csv").map { |row| row.to_h }
```

Algorithms

Pass an algorithm with:

```ruby
Eps::Model.new(data, algorithm: :linear_regression)
```

Eps supports:

- LightGBM (default)

- Linear Regression
- Naive Bayes

LightGBM

Pass the learning rate with:

```ruby
Eps::Model.new(data, learning_rate: 0.01)
```

Linear Regression

By default, an intercept is included. Disable this with:

```ruby
Eps::Model.new(data, intercept: false)
```

To speed up training on large datasets with linear regression, install GSL. With Homebrew, you can use:

```sh
brew install gsl
```

Then, add this line to your application's Gemfile:

```ruby
gem "gslr", group: :development
```

It only needs to be available in environments used to build the model.

Probability

To get the probability of each category for predictions with classification, use:

```ruby
model.predict_probability(data)
```

Naive Bayes is known to produce poor probability estimates, so stick with LightGBM if you need this.

Validation Options

Pass your own validation set with:

```ruby
Eps::Model.new(data, validation_set: validation_set)
```

Split on a specific value

```
Eps::Model.new(data, split: {column: :listed_at, value: Date.parse("
    2019-01-01")})
```

Specify the validation set size (the default is `0.25`, which is 25%)

```
Eps::Model.new(data, split: {validation_size: 0.2})
```

Disable the validation set completely with:

```
Eps::Model.new(data, split: false)
```

Database Storage

The database is another place you can store models. It's good if you retrain models automatically.

> We recommend adding monitoring and guardrails as well if you retrain automatically

Create an Active Record model to store the predictive model.

```
rails generate model Model key:string:uniq data:text
```

Store the model with:

```
store = Model.where(key: "price").first_or_initialize
store.update(data: model.to_pmml)
```

Load the model with:

```
data = Model.find_by!(key: "price").data
model = Eps::Model.load_pmml(data)
```

Jupyter & IRuby

You can use IRuby to run Eps in Jupyter notebooks. Here's how to get IRuby working with Rails.

Weights

Specify a weight for each data point

```
Eps::Model.new(data, weight: :weight)
```

You can also pass an array

```ruby
Eps::Model.new(data, weight: [1, 2, 3])
```

Weights are supported for metrics as well

```ruby
Eps.metrics(actual, predicted, weight: weight)
```

Reweighing is one method to mitigate bias in training data

Upgrading

0.3.0

Eps 0.3.0 brings a number of improvements, including support for LightGBM and cross-validation. There are a number of breaking changes to be aware of:

- LightGBM is now the default for new models. On Mac, run:

  ```sh
  brew install libomp
  ```

  Pass the `algorithm` option to use linear regression or naive Bayes.

  ```ruby
  Eps::Model.new(data, algorithm: :linear_regression) # or :
      naive_bayes
  ```

- Cross-validation happens automatically by default. You no longer need to create training and test sets manually. If you were splitting on a time, use:

  ```ruby
  Eps::Model.new(data, split: {column: :listed_at, value: Date.parse
      ("2019-01-01")})
  ```

  Or randomly, use:

  ```ruby
  Eps::Model.new(data, split: {validation_size: 0.3})
  ```

  To continue splitting manually, use:

  ```ruby
  Eps::Model.new(data, validation_set: test_set)
  ```

- It's no longer possible to load models in JSON or PFA formats. Retrain models and save them as PMML.

0.2.0

Eps 0.2.0 brings a number of improvements, including support for classification.

We recommend:

1. Changing `Eps::Regressor` to `Eps::Model`
2. Converting models from JSON to PMML

```ruby
model = Eps::Model.load_json("model.json")
File.write("model.pmml", model.to_pmml)
```

3. Renaming `app/stats_models` to `app/ml_models`

History

View the changelog

Contributing

Everyone is encouraged to help improve this project. Here are a few ways you can help:

- Report bugs
- Fix bugs and submit pull requests
- Write, clarify, or fix documentation
- Suggest or add new features

To get started with development:

```sh
git clone https://github.com/ankane/eps.git
cd eps
bundle install
bundle exec rake test
```