Multilayer Neural Network 2

Jony Sugianto
jony@evolvemachinelearners.com
0812-13086659
github.com/jonysugianto

Scikit Neuralnetwork

- Implementation of multilayer perceptron that compatible with scikit-learn
- pip install scikit-neuralnetwork
- pip install --upgrade https://github.com/Lasagne/Lasagne/archive/master.zip
  (optional)

Layer Specifications

class sknn.mlp.Layer(type, warning=None, name=None, units=None,
weight_decay=None, dropout=None,
normalize=None, frozen=False)

- type: str

  Select which activation function this layer should use, as a string. Specifically, options are *Rectifier, Sigmoid, Tanh, and ExpLin* for non-linear layers and *Linear* or *Softmax* for output layers.

- units: int

  The number of units (also known as neurons) in this layer. This applies to all layer types except for convolution.

MultiLayerPerceptron Specifications

class sknn.mlp.MultiLayerPerceptron(
layers, warning=None, parameters=None, random_state=None,
learning_rule=u'sgd', learning_rate=0.01, learning_momentum=0.9,
normalize=None, regularize=None, weight_decay=None,
dropout_rate=None, batch_size=1, n_iter=None, n_stable=10,
f_stable=0.001, valid_set=None, valid_size=0.0, loss_type=None,
callback=None, debug=False, verbose=None, **params)

- layers: list of Layer

  An iterable sequence of each layer each as a sknn.mlp.Layer instance that contains its type, optional name, and any paramaters required.

  - For hidden layers, you can use the following layer types: Rectifier, ExpLin, Sigmoid, Tanh, or Convolution.
  - For output layers, you can use the following layer types: Linear or Softmax.

- learning_rule: str, optional

  Name of the learning rule used during stochastic gradient descent, one of sgd, momentum, nesterov, adadelta, adagrad or rmsprop at the moment. The default is vanilla sgd.

MultiLayerPerceptron Specifications

- loss_type: string, optional

  The default option is *mse* for regressors and *mcc* for classifiers.

- learning_rate: float, optional

  Real number indicating the default/starting rate of adjustment for the weights during gradient descent. Different learning rules may take this into account differently. Default is 0.01.

- batch_size: int, optional

  Number of training samples to group together when performing stochastic gradient descent (technically, a “minibatch”). By default each sample is treated on its own, with batch_size=1. Larger batches are usually faster.

- n_iter: int, optional

  The number of iterations of gradient descent to perform on the neural network’s weights when training with fit().

Regressor Specifications

- fit(X, y, w=None)

  Fit the neural network to the given continuous data as a regression problem.

  - Parameters:
    - X : array-like, shape (n_samples, n_inputs)

      Training vectors as real numbers, where n_samples is the number of samples and n_inputs is the number of input features.
    - y : array-like, shape (n_samples, n_outputs)

      Target values are real numbers used as regression targets.
    - w : array-like (optional), shape (n_samples)

      Floating point weights for each of the training samples, used as mask to modify the cost function during optimization.
  - Returns: self : object

    Returns this instance.

Regressor Specifications

- predict(X)

  Calculate predictions for specified inputs.

  - Parameters: X : array-like, shape (n_samples, n_inputs)

    The input samples as real numbers.
  - Returns: y : array, shape (n_samples, n_outputs)

    The predicted values as real numbers.

Classifier Specifications

- fit(X, y, w=None)

  Fit the neural network to symbolic labels as a classification problem.

  - Parameters:
    - X : array-like, shape (n_samples, n_features)

      Training vectors as real numbers, where n_samples is the number of samples and n_inputs is the number of input features.
    - y : array-like, shape (n_samples, n_classes)

      Target values as integer symbols, for either single- or multi-output classification problems.
    - w : array-like (optional), shape (n_samples)

      Floating point weights for each of the training samples, used as mask to modify the cost function during optimization.

Classifier Specifications

- predict_proba(X, collapse=True)

  Calculate probability estimates based on these input features.

  - Parameters:
    - X : array-like of shape [n_samples, n_features]

      The input data as a numpy array.
  - Returns: y_prob : list of arrays of shape [n_samples, n_features, n_classes]

    The predicted probability of the sample for each class in the model, in the same order as the classes.

Dataset loader

```
from random import seed
import datetime
import random
import numpy as np

from csv import reader
```

Dataset loader

```
def check_valid_row(row):
    for c in row:
        if len(c.strip())==0:
            return False
    return True

# Load a CSV file
def load_csv(filename):
    dataset = list()
    with open(filename, 'r') as file:
        csv_reader = reader(file)
        for row in csv_reader:
            if not row:
                continue
            if check_valid_row(row):
                dataset.append(row)
    return dataset
```

Convert function

```
# Convert string column to float
def str_column_to_float(dataset, column):
	for row in dataset:
		row[column] = float(row[column].strip())

# Convert string column to integer
def str_column_to_int(dataset, column):
	class_values = [row[column] for row in dataset]
	unique = set(class_values)
	lookup = dict()
	for i, value in enumerate(unique):
		lookup[value] = i
	for row in dataset:
		row[column] = lookup[row[column]]
	return lookup
```

Test Dataset

```
if __name__ == "__main__":
    seed(datetime.datetime.utcnow())
    # load and prepare data
    filename = 'airfoil_self_noise.csv'
    dataset = load_csv(filename)
    for i in range(len(dataset[0])):
        str_column_to_float(dataset, i)
    print('transformed dataset')
    print(dataset)
```

Find min, max, mean dataset

```
# Find the min, max, mean values for each column
def dataset_minmaxmean(dataset):
	minmax=[]
	for col in range(len(dataset[0])):
		columnvalues=[]
		for row in range(len(dataset)):
			columnvalues.append(dataset[row][col])
		minmax.append([min(columnvalues), max(columnvalues), np.mean(columnvalues)])
	return minmax
```

Test Min, Max, Mean Dataset

```
if __name__ == "__main__":
	seed(datetime.datetime.utcnow())
	# load and prepare data
	filename = 'airfoil_self_noise.csv'
	dataset = load_csv(filename)
	for i in range(len(dataset[0])):
		str_column_to_float(dataset, i)
	print('transformed dataset')
	print(dataset)
	minmax = dataset_minmaxmean(dataset)
	print(minmax)
```

Normalize Dataset

```
# Rescale dataset columns to the range 0-1
def normalize_inputdataset(dataset, minmaxmean):
        for row in dataset:
                for col in range(len(row)-1):
                        range_value=(minmaxmean[col][1] - minmaxmean[col][0])
                        row[col] = (row[col] - minmaxmean[col][0]) / range_value
```

Test Normalize Dataset

```
if __name__ == "__main__":
	seed(datetime.datetime.utcnow())
	# load and prepare data
	filename = '/data/presentation/eml/python/src/week6/airfoil_self_noise.csv'
	dataset = load_csv(filename)
	for i in range(len(dataset[0])):
		str_column_to_float(dataset, i)
	print('transformed dataset')
	print(dataset)
	minmax = dataset_minmaxmean(dataset)
	print(minmax)
	normalize_inputdataset(dataset, minmax)
	print(dataset)
```

Split dataset into trainset and testset

```
def split_into_training_test_set(dataset, split):
        trainingSet = []
        testSet = []
        for d in dataset:
                if random.random() < split:
                        trainingSet.append(d)
                else:
                        testSet.append(d)

        return trainingSet, testSet
```

Test Split dataset

```
if __name__ == "__main__":
	seed(datetime.datetime.utcnow())
	# load and prepare data
	filename = '/data/presentation/eml/python/src/week6/airfoil_self_noise.csv'
	dataset = load_csv(filename)
	for i in range(len(dataset[0])):
		str_column_to_float(dataset, i)
	print('transformed dataset')
	print(dataset[0])
	minmax = dataset_minmaxmean(dataset)
	print('min, max, mean')
	print(minmax)
	normalize_inputdataset(dataset, minmax)
	print('normalize dataset')
	print(dataset[0])
	split=0.8
	trainingset, testset=split_into_training_test_set(dataset, split)
	print('size trainingset', len(trainingset), 'size testset', len(testset))
```

Split dataset into x and y

```
def split_into_x_and_y(data):
    x=map(lambda item: item[:len(item)-1], data)
    y=map(lambda item: [item[len(item)-1]], data)
    return np.array(list(x)), np.array(list(y))
```

Test Split dataset into x and y

```
if __name__ == "__main__":
    seed(datetime.datetime.utcnow())
    # load and prepare data
    filename = '/data/presentation/eml/python/src/week6/airfoil_self_noise.csv'
    dataset = load_csv(filename)
    for i in range(len(dataset[0])):
        str_column_to_float(dataset, i)
    print('transformed dataset')
    print(dataset[0])
    minmax = dataset_minmaxmean(dataset)
    print('min, max, mean')
    print(minmax)
    normalize_inputdataset(dataset, minmax)
    print('normalize dataset')
    print(dataset[0])
    split=0.8
    trainingset, testset=split_into_training_test_set(dataset, split)
    print('size trainingset', len(trainingset), 'size testset', len(testset))
    train_x, train_y=split_into_x_and_y(trainingset)
    print('train x',train_x[0])
    print('train y',train_y[0])
```

Wheat varieties classification

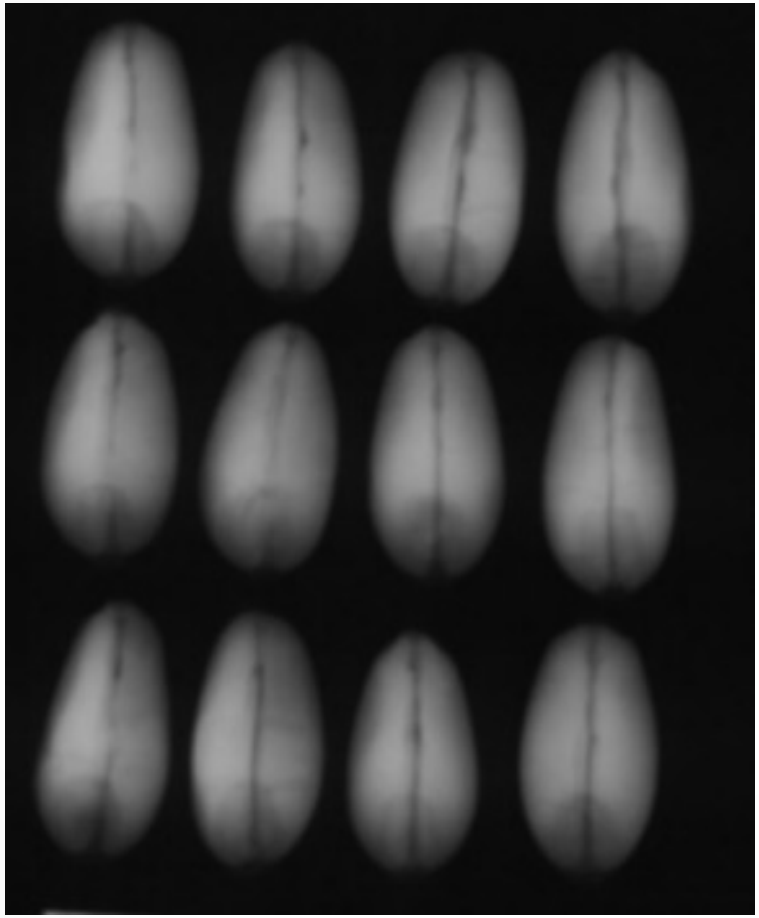
EVOLVE
MACHINE LEARNERS

Wheat varieties classification

- Kama
- Rosa
- Canadian

Wheat classification

X-ray Photograms



Computing Features

To construct the data, seven geometric parameters of wheat kernels were measured:

- area A,
- perimeter P,
- compactness C = 4*pi*A/P^2,
- length of kernel,
- width of kernel,
- asymmetry coefficient
- length of kernel groove.

Sample data

15.26,14.84,0.871,5.763,3.312,2.221,5.22,1
14.88,14.57,0.8811,5.554,3.333,1.018,4.956,1
14.29,14.09,0.905,5.291,3.337,2.699,4.825,1
13.84,13.94,0.8955,5.324,3.379,2.259,4.805,1
16.14,14.99,0.9034,5.658,3.562,1.355,5.175,1

Import library

```
from random import seed
from week6.csv_dataset import load_csv, str_column_to_float,
str_column_to_int
from week6.csv_dataset import dataset_minmaxmean, normalize_inputdataset
from week6.csv_dataset import split_into_training_test_set, split_into_x_and_y
from sknn.mlp import Classifier, Layer
import datetime
```

Create network

```
def create_network(niter, lr,
verboseflag):
    nn = Classifier(
        layers=[
            Layer("Sigmoid", units=5),
            Layer("Softmax")],
        learning_rate=lr,
        n_iter=niter, verbose=verboseflag)

    return nn
```

Compute accuracy

```
# Calculate accuracy percentage
def accuracy_metric(actual, predicted):
	correct = 0
	for i in range(len(actual)):
		if actual[i] == predicted[i]:
			correct += 1
	return correct / float(len(actual)) * 100.0
```

Wheat classification without normalization

EVOLVE
MACHINE LEARNERS

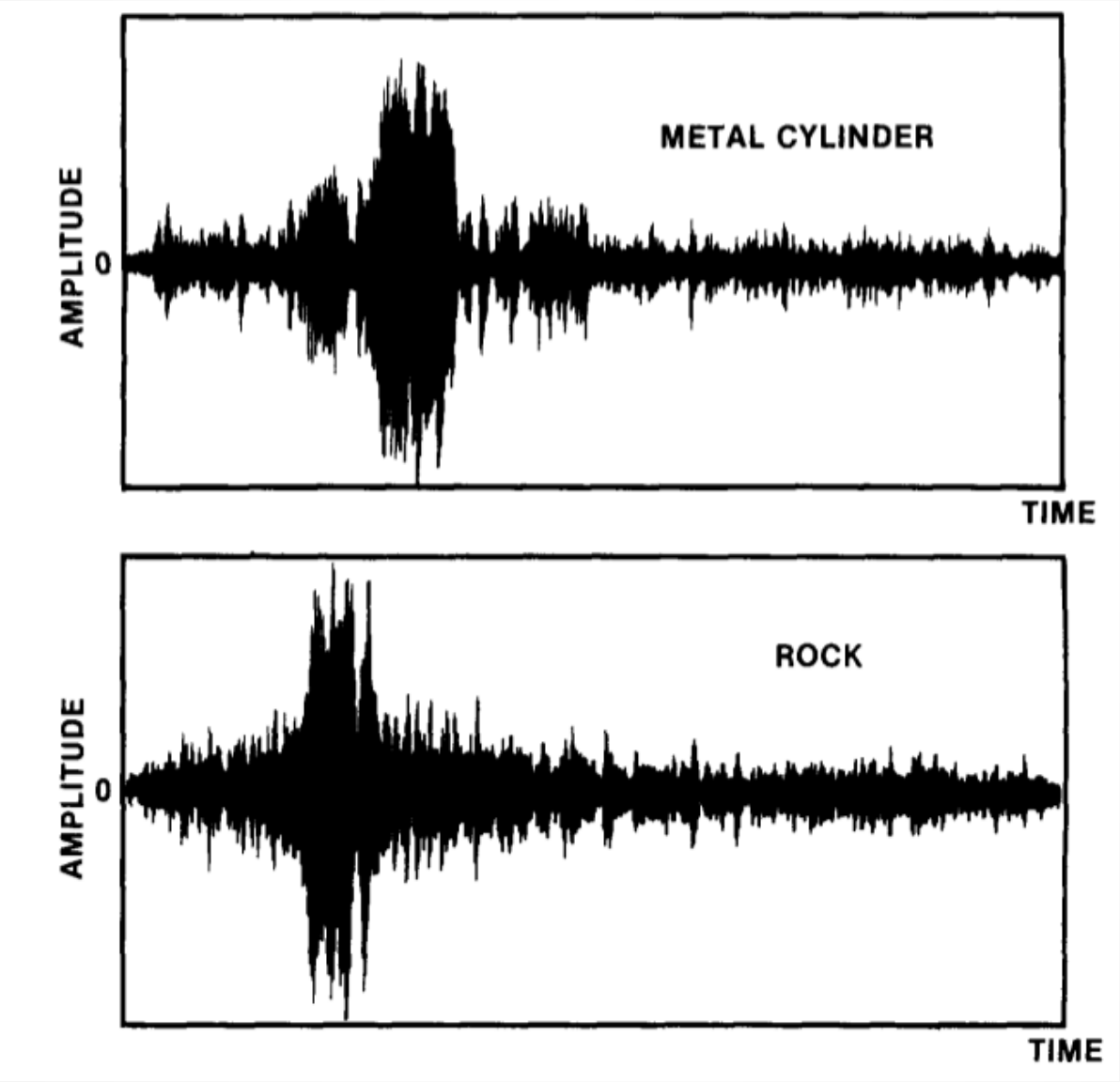
```
if __name__ == "__main__":
    seed(datetime.datetime.utcnow())
    filename = '/data/presentation/eml/python/src/week6/wheat-seeds.csv'
    dataset = load_csv(filename)
    for i in range(len(dataset[0]) - 1):
        str_column_to_float(dataset, i)
    str_column_to_int(dataset, len(dataset[0]) - 1)
    minmax = dataset_minmaxmean(dataset)
    print('stats dataset', minmax)
    #normalize_inputdataset(dataset, minmax)
    split=0.8
    trainingset, testset = split_into_training_test_set(dataset, split)
    train_x, train_y = split_into_x_and_y(trainingset)
    niter=100
    lr=0.01
    verboseflag=True
    model=create_network(niter, lr, verboseflag)
    model.fit(train_x, train_y)
    predicted_train_y=model.predict(train_x)
    acc_train=accuracy_metric(train_y, predicted_train_y)
    print('accuracy on training set', acc_train)
    test_x, test_y = split_into_x_and_y(testset)
    predicted_test_y=model.predict(test_x)
    acc_test = accuracy_metric(test_y, predicted_test_y)
    print('accuracy on test set', acc_test)
```

Wheat classification with normalization

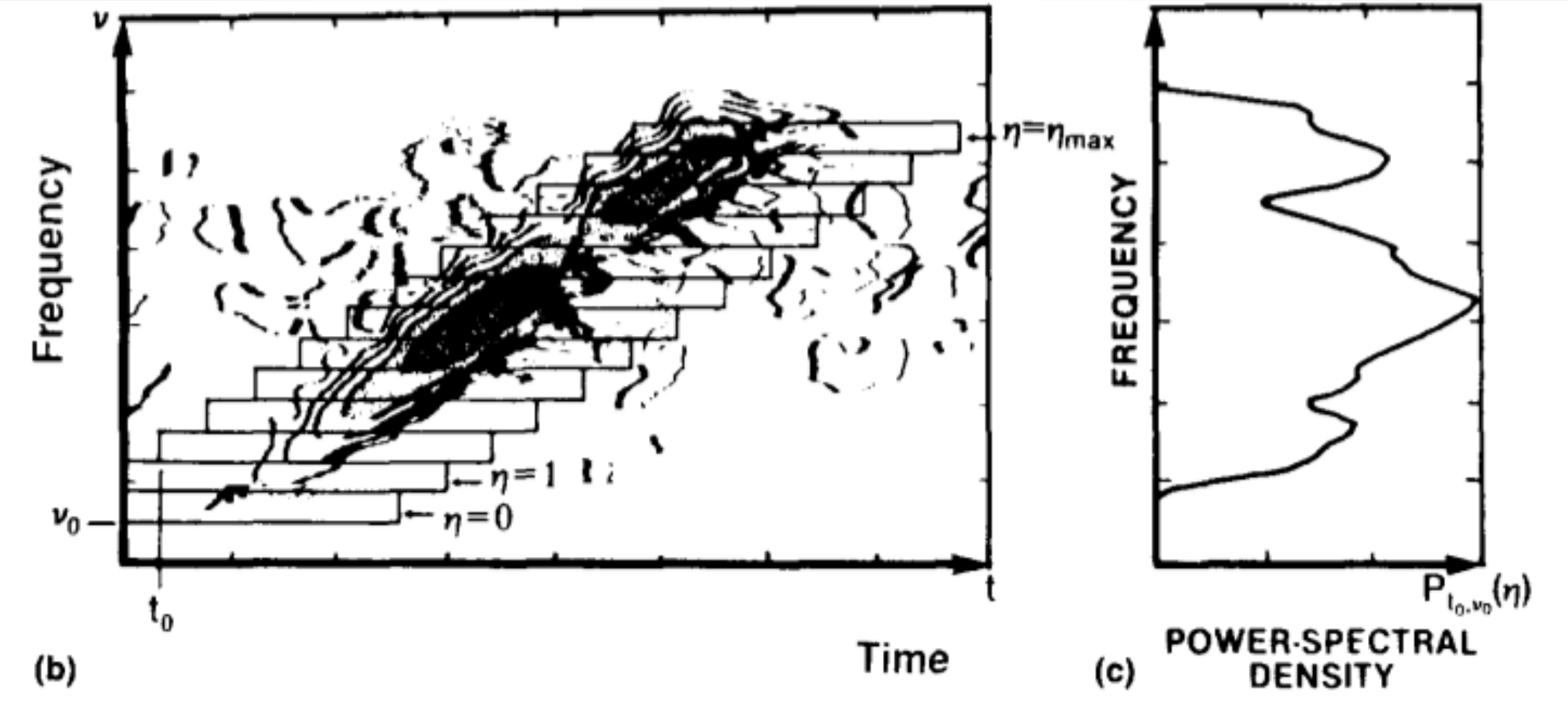
```
if __name__ == "__main__":
    seed(datetime.datetime.utcnow())
    filename = '/data/presentation/eml/python/src/week6/wheat-seeds.csv'
    dataset = load_csv(filename)
    for i in range(len(dataset[0]) - 1):
        str_column_to_float(dataset, i)
    str_column_to_int(dataset, len(dataset[0]) - 1)
    minmax = dataset_minmaxmean(dataset)
    print('stats dataset', minmax)
    normalize_inputdataset(dataset, minmax)
    split=0.8
    trainingset, testset = split_into_training_test_set(dataset, split)
    train_x, train_y = split_into_x_and_y(trainingset)
    niter=100
    lr=0.01
    verboseflag=True
    model=create_network(niter, lr, verboseflag)
    model.fit(train_x, train_y)
    predicted_train_y=model.predict(train_x)
    acc_train=accuracy_metric(train_y, predicted_train_y)
    print('accuracy on training set', acc_train)
    test_x, test_y = split_into_x_and_y(testset)
    predicted_test_y=model.predict(test_x)
    acc_test = accuracy_metric(test_y, predicted_test_y)
    print('accuracy on test set', acc_test)
```

undersea metal cylinder and a cylindrically shaped rock Classification

Sonar Images



Preprocessing Sonar Images



Sonar Dataset

0.0291,0.0400,0.0771,0.0809,0.0521,0.1051,0.0145,0.0674,0.1294,0.1146,0.0942,0.0794
,0.0252,0.1191,0.1045,0.2050,0.1556,0.2690,0.3784,0.4024,0.3470,0.1395,0.1208,0.282
7,0.1500,0.2626,0.4468,0.7520,0.9036,0.7812,0.4766,0.2483,0.5372,0.6279,0.3647,0.45
72,0.6359,0.6474,0.5520,0.3253,0.2292,0.0653,0.0000,0.0000,0.0000,0.0000,0.0000,0.0
000,0.0000,0.0000,0.0000,0.0056,0.0237,0.0204,0.0050,0.0137,0.0164,0.0081,0.0139,0.
0111,R
0.0181,0.0146,0.0026,0.0141,0.0421,0.0473,0.0361,0.0741,0.1398,0.1045,0.0904,0.0671
,0.0997,0.1056,0.0346,0.1231,0.1626,0.3652,0.3262,0.2995,0.2109,0.2104,0.2085,0.228
2,0.0747,0.1969,0.4086,0.6385,0.7970,0.7508,0.5517,0.2214,0.4672,0.4479,0.2297,0.32
35,0.4480,0.5581,0.6520,0.5354,0.2478,0.2268,0.1788,0.0898,0.0536,0.0374,0.0990,0.0
956,0.0317,0.0142,0.0076,0.0223,0.0255,0.0145,0.0233,0.0041,0.0018,0.0048,0.0089,0.
0085,R
0.0491,0.0279,0.0592,0.1270,0.1772,0.1908,0.2217,0.0768,0.1246,0.2028,0.0947,0.2497
,0.2209,0.3195,0.3340,0.3323,0.2780,0.2975,0.2948,0.1729,0.3264,0.3834,0.3523,0.541
0,0.5228,0.4475,0.5340,0.5323,0.3907,0.3456,0.4091,0.4639,0.5580,0.5727,0.6355,0.75
63,0.6903,0.6176,0.5379,0.5622,0.6508,0.4797,0.3736,0.2804,0.1982,0.2438,0.1789,0.1
706,0.0762,0.0238,0.0268,0.0081,0.0129,0.0161,0.0063,0.0119,0.0194,0.0140,0.0332,0.
0439,M

Import Dataset

```
from random import seed
from week6.csv_dataset import load_csv, str_column_to_float,
str_column_to_int
from week6.csv_dataset import split_into_training_test_set, split_into_x_and_y
from week6.csv_dataset import normalize_inputdataset, dataset_minmaxmean
from sknn.mlp import Classifier, Layer
import datetime
```

Create network

```
def create_network(niter, lr,
verboseflag):
    nn = Classifier(
        layers=[
            Layer("Sigmoid", units=10),
            Layer("Softmax")],
        learning_rate=lr,
        n_iter=niter, verbose=verboseflag)

    return nn
```

Compute accuracy

EVOLVE
MACHINE LEARNERS

```
# Calculate accuracy percentage
def accuracy_metric(actual, predicted):
	correct = 0
	for i in range(len(actual)):
		if actual[i] == predicted[i]:
			correct += 1
	return correct / float(len(actual)) * 100.0
```

Sonar classification without normalization

```
if __name__ == "__main__":
    seed(datetime.datetime.utcnow())
    filename = '/data/presentation/eml/python/src/week6/sonar.all-
data.csv'
    dataset = load_csv(filename)
    for i in range(len(dataset[0]) - 1):
        str_column_to_float(dataset, i)
    str_column_to_int(dataset, len(dataset[0]) - 1)
    minmax = dataset_minmaxmean(dataset)
    print('stats dataset', minmax)
    #normalize_inputdataset(dataset, minmax)
    split=0.8
    trainingset, testset = split_into_training_test_set(dataset, split)
    train_x, train_y = split_into_x_and_y(trainingset)
    niter=100
    lr=0.01
    verboseflag=True
    model=create_network(niter, lr, verboseflag)
    model.fit(train_x, train_y)
    predicted_train_y=model.predict(train_x)
    acc_train=accuracy_metric(train_y, predicted_train_y)
    print('accuracy on training set', acc_train)
    test_x, test_y = split_into_x_and_y(testset)
    predicted_test_y=model.predict(test_x)
    acc_test = accuracy_metric(test_y, predicted_test_y)
    print('accuracy on test set', acc_test)
```

Sonar classification with normalization

```
if __name__ == "__main__":
    seed(datetime.datetime.utcnow())
    filename = '/data/presentation/eml/python/src/week6/sonar.all-
data.csv'
    dataset = load_csv(filename)
    for i in range(len(dataset[0]) - 1):
        str_column_to_float(dataset, i)
    str_column_to_int(dataset, len(dataset[0]) - 1)
    minmax = dataset_minmaxmean(dataset)
    print('stats dataset', minmax)
    normalize_inputdataset(dataset, minmax)
    split=0.8
    trainingset, testset = split_into_training_test_set(dataset, split)
    train_x, train_y = split_into_x_and_y(trainingset)
    niter=100
    lr=0.01
    verboseflag=True
    model=create_network(niter, lr, verboseflag)
    model.fit(train_x, train_y)
    predicted_train_y=model.predict(train_x)
    acc_train=accuracy_metric(train_y, predicted_train_y)
    print('accuracy on training set', acc_train)
    test_x, test_y = split_into_x_and_y(testset)
    predicted_test_y=model.predict(test_x)
    acc_test = accuracy_metric(test_y, predicted_test_y)
    print('accuracy on test set', acc_test)
```

Airfoil self-noise prediction

Airfoil dataset

- **frequency**, in Hertzs, used as input.
- **angle_of_attack**, in degrees, used as input.
- **chord_length**, in meters, used as input.
- **free_stream_velocity**, in meters per second, used as input.
- **suction_side_displacement_thickness**, in meters, used as input.
- **scaled_sound_pressure_level**, in decibels, used as target.

Airfoil dataset statistics

	Min.	Max.	Mean	Std.
frequency	200	2e+004	2.89e+003	3.15e+003
angle_of_attack	0	22.2	6.78	5.92
chord_length	0.0254	0.305	0.137	0.0935
free_stream_velocity	31.7	71.3	50.9	15.6
displacement_thickness	0.000401	0.0584	0.0111	0.0132
sound_level	103	141	125	6.9

Import library

```
from random import seed
from week6.csv_dataset import load_csv, str_column_to_float
from week6.csv_dataset import dataset_minmaxmean, normalize_inputdataset
from week6.csv_dataset import split_into_training_test_set, split_into_x_and_y
from sknn.mlp import Regressor, Layer
import numpy as np
import datetime
```

Create network

```
def create_network(niter, lr, vf):
    nn = Regressor(
        layers=[
            Layer("Sigmoid", units=5),
            Layer("Linear")],
        learning_rate=lr,
        n_iter=niter, verbose=vf)

    return nn
```

Model evaluation

```
# Model Evaluation - RMSE
def rmse_score(Y, Y_pred):
    y_min_y_pred=Y - Y_pred
    rmse = np.sqrt(sum(y_min_y_pred ** 2) / float(len(Y)))
    return rmse

# Model Evaluation - R2 Score
def r2_score(Y, Y_pred):
    mean_y = np.mean(Y)
    ss_tot = sum((Y - mean_y) ** 2)
    ss_res = sum((Y - Y_pred) ** 2)
    r2 = 1 - (ss_res / ss_tot)
    return r2
```

Airfoil prediction without normalization

```
if __name__ == "__main__":
    seed(datetime.datetime.utcnow())
    filename = '/data/presentation/eml/python/src/week6/airfoil_self_noise.csv'
    dataset = load_csv(filename)
    for i in range(len(dataset[0]) ):
        str_column_to_float(dataset, i)
    minmax = dataset_minmaxmean(dataset)
    print('stats dataset', minmax)
    #normalize_inputdataset(dataset, minmax)
    split=0.7
    trainingset, testset = split_into_training_test_set(dataset, split)
    train_x, train_y = split_into_x_and_y(trainingset)
    niter=100
    lr=0.01
    vf=True
    model=create_network(niter, lr, vf)
    model.fit(train_x, train_y)
    predicted_train_y=model.predict(train_x)
    rmse = rmse_score(train_y, predicted_train_y)
    r2 = r2_score(train_y, predicted_train_y)
    print("RMSE trainingset", rmse, "R2 Score trainingset", r2)
    test_x, test_y = split_into_x_and_y(testset)
    predicted_test_y = model.predict(test_x)
    rmse = rmse_score(test_y, predicted_test_y)
    r2 = r2_score(test_y, predicted_test_y)
    print("RMSE testset", rmse,"R2 Score testset", r2)
```

Airfoil prediction with normalization

```
if __name__ == "__main__":
    seed(datetime.datetime.utcnow())
    filename = '/data/presentation/eml/python/src/week6/airfoil_self_noise.csv'
    dataset = load_csv(filename)
    for i in range(len(dataset[0]) ):
        str_column_to_float(dataset, i)
    minmax = dataset_minmaxmean(dataset)
    print('stats dataset', minmax)
    normalize_inputdataset(dataset, minmax)
    split=0.7
    trainingset, testset = split_into_training_test_set(dataset, split)
    train_x, train_y = split_into_x_and_y(trainingset)
    niter=100
    lr=0.01
    vf=True
    model=create_network(niter, lr, vf)
    model.fit(train_x, train_y)
    predicted_train_y=model.predict(train_x)
    rmse = rmse_score(train_y, predicted_train_y)
    r2 = r2_score(train_y, predicted_train_y)
    print("RMSE trainingset", rmse, "R2 Score trainingset", r2)
    test_x, test_y = split_into_x_and_y(testset)
    predicted_test_y = model.predict(test_x)
    rmse = rmse_score(test_y, predicted_test_y)
    r2 = r2_score(test_y, predicted_test_y)
    print("RMSE testset", rmse,"R2 Score testset", r2)
```

Improving Backprop Performance

- Avoiding local minima
- Keep derivatives from going to zero
- For classifiers, use reachable targets
- Compensate for error attenuation in deep layers
- Compensate for fan-in effects
- Use momentum to speed learning
- Reduce learning rate when weights oscillate
- Use small initial random weights and small initial learning rate to avoid “herd effect”
- Cross-entropy error measure

Avoiding Local Minima

One problem with backprop is that the error surface is no longer bowl-shaped.

Gradient descent can get trapped in local minima.

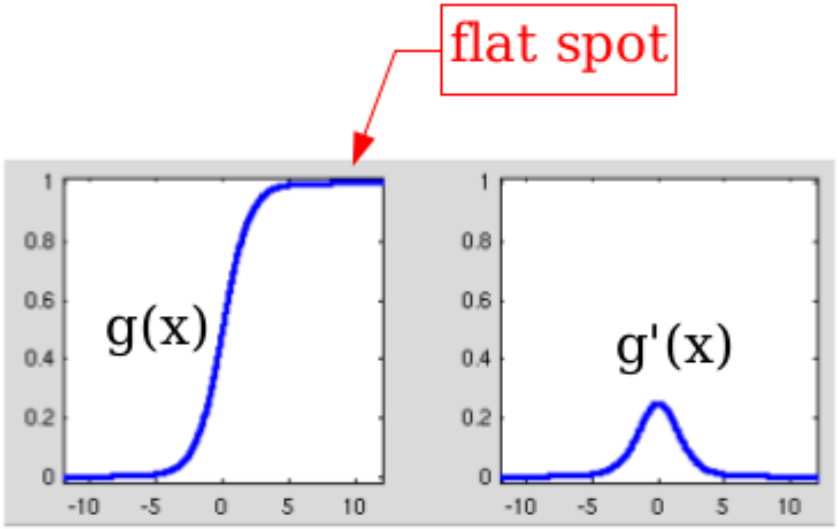
- In practice, this does not usually prevent learning.

"Noise" can get us out of local minima:

- Stochastic update (one pattern at a time).
- Add noise to training data, weights, or activations.
- Large learning rates can be a source of noise due to overshooting.

Flat Spots

If weights become large, net_j becomes large, derivative of g() goes to zero.



Fahlman's trick: add a small constant to g'(x) to keep the derivative from going to zero. Typical value is 0.1.

Reachable Targets for Classifiers

Targets of 0 and 1 are unreachable by the logistic or tanh functions.

Weights get large as the algorithm tries to force each output unit to reach its asymptotic value.

Trying to get a “correct” output from 0.95 up to 1.0 wastes time and resources that should be concentrated elsewhere.

Solution: use “reachable targets” of 0.1 and 0.9 instead of 0/1. And don't penalize the network for overshooting these targets.

Error Signal Attenuation

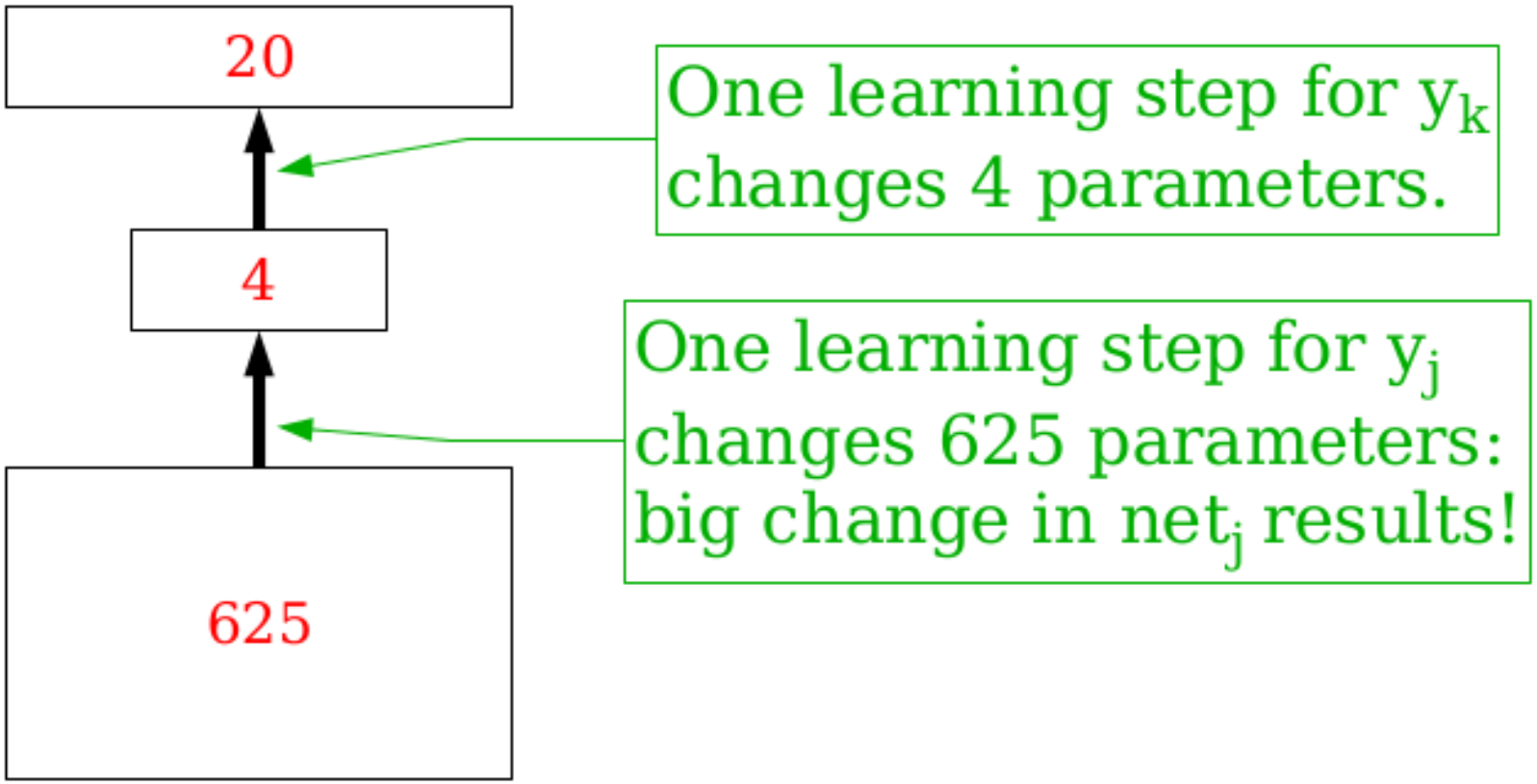
The error signal δ gets attenuated as it moves backward through multiple layers.

So different layers learn at different rates.

Input-to-hidden weights learn more slowly than hidden-to-output weights.

Solution: have different learning rates η for different layers.

Fan-In Affects Learning Rate



Solution: scale learning rate by fan-in.

Momentum

Learning is slow if the learning rate is set too low.

Gradient may be steep in some directions but shallow in others.

Solution: add a momentum term α.

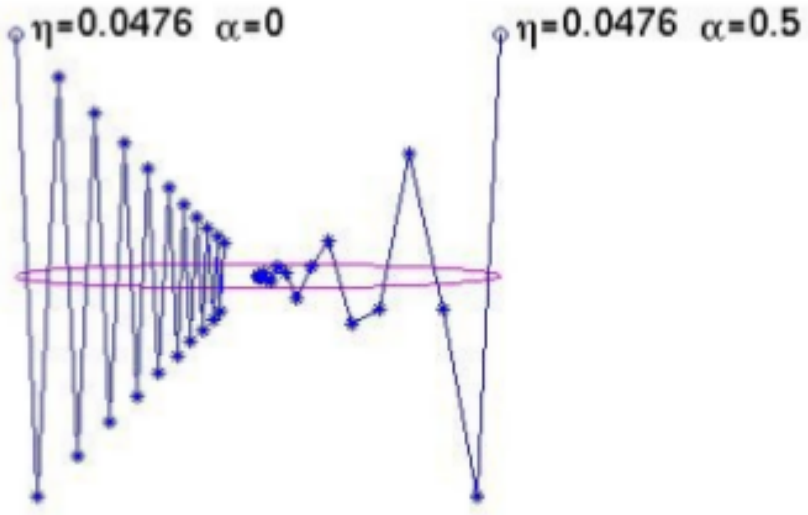
$$\Delta w_{ij}(t) = -\eta \frac{\partial E}{\partial w_{ij}(t)} + \alpha \cdot \Delta w_{ij}(t-1)$$

Typical value for α is 0.5.

If the direction of the gradient remains constant, the algorithm will take increasingly large steps.

Momentum Demo

Hertz, Krogh & Palmer figs. 5.10 and 6.3: gradient descent on a quadratic error surface E (no neural net) involved:



$$E = x^2 + 20y^2$$

$$\frac{\partial E}{\partial x} = 2x, \quad \frac{\partial E}{\partial y} = 40y$$

Initial $[x, y]=[-1,1]$ or $[1,1]$

Weights Can Oscillate If Learning Rate Set Too High

Solution: calculate the cosine of the angle between successive weight vectors.

$$\cos\theta = \frac{\vec{\Delta} w(t) \cdot \vec{\Delta} w(t-1)}{\|\vec{\Delta} w(t)\| \cdot \|\vec{\Delta} w(t-1)\|}$$

If cosine close to 1, things are going well.

If cosine < 0.95, reduce the learning rate.

If cosine < 0, we're oscillating: cancel the momentum.

$$\Delta w(t) = -\eta \frac{\partial E}{\partial w} + \alpha \cdot \Delta w(t-1)$$

Cross-Entropy Error Measure

- Alternative to sum-squared error for binary outputs; diverges when the network gets an output completely wrong.

$$E = \sum_p \left[d^p \log \frac{d^p}{y^p} + (1-d^p) \log \frac{1-d^p}{1-y^p} \right]$$

- Can produce faster learning for some types of problems.
- Can learn some problems where sum-squared error gets stuck in a local minimum, because it heavily penalizes “very wrong” outputs.

Evaluation Metric ML algorithm

- Accuracy
- Precision
- Recall

Evaluation Metric ML algorithm

		Predicted class	
Actual Class		Class = Yes	Class = No
	Class = Yes	True Positive	False Negative
	Class = No	False Positive	True Negative

- **True Positives (TP) -** These are the correctly predicted positive values which means that the value of actual class is yes and the value of predicted class is also yes. E.g. if actual class value indicates that this passenger survived and predicted class tells you the same thing.
- **True Negatives (TN) -** These are the correctly predicted negative values which means that the value of actual class is no and value of predicted class is also no. E.g. if actual class says this passenger did not survive and predicted class tells you the same thing.
- **False Positives (FP)** – When actual class is no and predicted class is yes. E.g. if actual class says this passenger did not survive but predicted class tells you that this passenger will survive.
- **False Negatives (FN)** – When actual class is yes but predicted class in no. E.g. if actual class value indicates that this passenger survived and predicted class tells you that passenger will die.

Accuracy

Accuracy is the most intuitive performance measure and it is simply a ratio of correctly predicted observation to the total observations. One may think that, if we have high accuracy then our model is best. Yes, accuracy is a great measure but only when you have symmetric datasets where values of false positive and false negatives are almost same. Therefore, you have to look at other parameters to evaluate the performance of your model. For our model, we have got 0.803 which means our model is approx. 80% accurate.

Accuracy = (TP+TN)/(TP+FP+FN+TN)

Precision

Precision is the ratio of correctly predicted positive observations to the total predicted positive observations. The question that this metric answer is of all passengers that labeled as survived, how many actually survived? High precision relates to the low false positive rate. We have got 0.788 precision which is pretty good.

Precision = TP/(TP+FP)

Recall/Sensitivity

Recall is the ratio of correctly predicted positive observations to the all observations in actual class - yes. The question recall answers is: Of all the passengers that truly survived, how many did we label? We have got recall of 0.631 which is good for this model as it's above 0.5.

Recall = TP/TP+FN

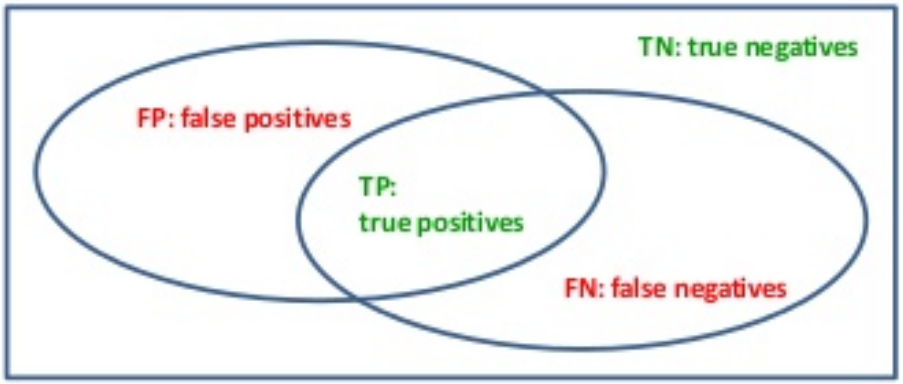
F1 score

F1 score - F1 Score is the weighted average of Precision and Recall. Therefore, this score takes both false positives and false negatives into account. Intuitively it is not as easy to understand as accuracy, but F1 is usually more useful than accuracy, especially if you have an uneven class distribution. Accuracy works best if false positives and false negatives have similar cost. If the cost of false positives and false negatives are very different, it's better to look at both Precision and Recall. In our case, F1 score is 0.701.

F1 Score = 2*(Recall * Precision) / (Recall + Precision)

Summary Evaluation Metric

Accuracy, Precision, Recall, and F-measure



Precision:

$$p = \frac{TP}{TP + FP}$$

Recall:

$$r = \frac{TP}{TP + FN}$$

Accuracy:

$$acc = \frac{TP + TN}{TP + TN + FP + FN}$$

F-measure: Harmonic mean of precision and recall

$$F = \frac{1}{\frac{1}{2}\left(\frac{1}{p} + \frac{1}{r}\right)} = \frac{2pr}{p + r}$$

Summary Evaluation Metric