drifter_ml Documentation

Release 0.20

Eric Schles

May 11, 2021

Contents

CHAPTER 1

Introduction

Welcome to Drifter, a tool to help you test your machine learning models. This testing framework is broken out semantically, so you can test different aspects of your machine learning system.

The tests come in two general flavors, component tests, like this one that tests for a minimum precision per class:

```
from drifter_ml.classification_tests import ClassificationTests
import joblib
import pandas as pd

def test_precision():
        clf = joblib.load("random_forest.joblib")
        test_data = pd.read_csv("test.csv")
        columns = test_data.columns.tolist()
        columns.remove("target")
        clf_tests = ClassificationTests(clf, test_data, "target", columns)
        classes = set(test_data["target"])
        precision_per_class = {klass: 0.9 for klass in classes}
        clf_tests.precision_lower_boundary_per_class(precision_per_class)
```

And an entire test suite that tests for precision, recall and f1 score in one test:

```
from drifter_ml.classification_tests import ClassificationTests
import joblib
import pandas as pd

def test_precision():
        clf = joblib.load("random_forest.joblib")
        test_data = pd.read_csv("test.csv")
        columns = test_data.columns.tolist()
        columns.remove("target")
        clf_tests = ClassificationTests(clf, test_data, "target", columns)
        classes = set(test_data["target"])
        precision_per_class = {klass: 0.9 for klass in classes}
        recall_per_class = {klass: 0.9 for klass in classes}
```

(continues on next page)

(continued from previous page)

```
        f1_per_class = {klass: 0.9 for klass in classes}
        clf_tests.classifier_testing(
        precision_per_class,
        recall_per_class,
        f1_per_class
        )
```

The expectation at present is that all models follow the scikit learn api, which means there is an expectation of a *fit* and *predict* on all models. This may appear exclusionary, but you can infact wrap keras models with scikit-learn style objects, allowing for the same api:

```
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import cross_val_score
import numpy

# Function to create model, required for KerasClassifier
def create_model():
        # create model
        model = Sequential()
        model.add(Dense(12, input_dim=8, activation='relu'))
        model.add(Dense(8, activation='relu'))
        model.add(Dense(1, activation='sigmoid'))
        # Compile model
        model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy
↪'])
        return model

# fix random seed for reproducibility
seed = 7
numpy.random.seed(seed)
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
# create model
model = KerasClassifier(build_fn=create_model, epochs=150, batch_size=10, verbose=0)
# evaluate using 10-fold cross validation
kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=seed)
results = cross_val_score(model, X, Y, cv=kfold)
print(results.mean())
```

This means that traditional machine learning and deep learning are available for testing out of the box!

CHAPTER 2

Project Setup

2.1 Regression and Classification Tests

If you are going to use regression or classification tests, you'll need to do a bit of setup. The first step is making sure you have a test set with labeled data that you can trust. It is recommended that you break your initial labeled dataset up into test and train and keep the test for both the model generation phase as well as for model monitoring throughout.

A good rule of thumb is to have 70% train, and 30% test. Other splits may be ideal, depending on the needs of your project. You can setup test and train using existing tools from sklearn as follows:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_
↪state=1)
```

Once you have your two datasets you can train your model with the training set, as is typical:

```
from sklearn import tree
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
import joblib

df = pd.DataFrame()
for _ in range(5000):
    a = np.random.normal(0, 1)
    b = np.random.normal(0, 3)
    c = np.random.normal(12, 4)
    if a + b + c > 11:
        target = 1
    else:
        target = 0
    df = df.append({
        "A": a,
        "B": b,
```

(continues on next page)

(continued from previous page)

```
        "C": c,
        "target": target
    }, ignore_index=True)

clf = tree.DecisionTreeClassifier()
X = df[["A", "B", "C"]]
y = df["target"]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_
↪state=1)

clf.fit(X_train, y_train)
joblib.dump(clf, "model.joblib")
df.to_csv("data.csv")
test_data = pd.DataFrame()
test_data[["A", "B", "C"]]
test_data["target"] = y_test
test_data.to_csv("test_data.csv")
```

Then you can test against your model before you put it into production as follows:

```
import joblib
import pandas as pd
from sklearn.metrics import f1_score

clf = joblib.load("model.joblib")
test_data = pd.read_csv("test_data.csv")
y_pred = clf.predict(test_data[["A", "B", "C"]])
y_true = test_data["target"]
print(f1_score(y_true, y_pred))
```

It's worth noting that one score is likely never good enough, you need to include multiple measures to ensure your model is not simply fitting towards a single measure. Assuming the measures are good enough you can move onto productionizing your model.

2.2 Strategies For Testing Your Productionized Model

Once you've put your model into production there are a few strategies for making sure your model continues to meet your requirements:

1. Using the test set from training - Gather new data and predictions from production and then training a new classifier or regressor with the new data and new predictions. Then test against the test set you've set aside. If the measures stay approximately the same, it's possible your model is performing as expected. It's important that the new classifier have the same hyper parameters as the one in production as well as using the same versions for all associated code that creates the new model object.
2. Generating a new test set from a process - Gather new data and new predictions from the production model. Then manually label the same set of new data, either via some people process or other process you believe to be able to generate faithful labels. Then validate that the manually labeled examples against the predicted examples. If you are predicting new data a lot, I recommend taking random non-overlapping samples from the production data and labeling those.
3. Generating a new test set from a process and then do label propagation - Gather new data and new predictions from the production model. Then manually label a small set of the new data in some manor. Make sure to have multiple people manually label the same data, till everyone agrees on the ground truth. Then generate a new set of labels via label propagation. Then have people manually label the newly propagated labels, if

the newly propagated labels agree with the manual labels often enough, then continue the label propagation process. Continue to check random non-overlapping samples until you feel satisified, then label the remainder of the production data.

2.3 Using The Test Set From Training

So the above description is a bit terse so let's break it down with some example code to inform your own project setup. First let's assume that you have some data to train on and test on:

```
from sklearn import tree
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
import joblib

df = pd.DataFrame()
for _ in range(5000):
    a = np.random.normal(0, 1)
    b = np.random.normal(0, 3)
    c = np.random.normal(12, 4)
    if a + b + c > 11:
        target = 1
    else:
        target = 0
    df = df.append({
        "A": a,
        "B": b,
        "C": c,
        "target": target
    }, ignore_index=True)

clf = tree.DecisionTreeClassifier()
X = df[["A", "B", "C"]]
y = df["target"]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_
↪state=1)

clf.fit(X_train, y_train)
joblib.dump(clf, "model.joblib")
df.to_csv("data.csv")
test_data = pd.DataFrame()
test_data[["A", "B", "C"]] = df[["A", "B", "C"]]
test_data["target"] = y_test
test_data.to_csv("test_data.csv")
```

Next we need to test our model to make sure it's performing well enough to go into production:

```
import joblib
import pandas as pd
from sklearn.metrics import classification_report

clf = joblib.load("model.joblib")
test_data = pd.read_csv("test_data.csv")
y_pred = clf.predict(test_data[["A", "B", "C"]])
y_true = test_data["target"]
print(classification_report(y_true, y_pred))
```

Let's assume everything met our minimum criteria for going to production. Now we are ready to put our model into production!! For this we'll need to write our test such that it makes use of the test data, our new data and our new predictions. For the purposes of the below example, assume you've been saving new data and new predictions to a csv called new_data.csv, that you have saved your production model in a file called model.joblib and that you have test data saved to test_data.csv. Below is an example test you might write using the framework:

```
import joblib
import pandas as pd
from sklearn import tree
from drifter_ml import classification_tests

def generate_model_from_production_data():
    new_data = pd.read_csv("new_data.csv")
    prod_clf = joblib.load("model.joblib")
    test_data = pd.read_csv("test_data.csv")
    return test_data, new_data, prod_clf

def test_precision():
    test_data, new_data, prod_clf = generate_model_from_production_data()
    column_names = ["A", "B", "C"]
    target_name = "target"
    test_clf = tree.DecisionTreeClassifier()
    test_clf.set_params(**prod_clf.get_params())
    X = new_data[column_names]
    y = new_data[target_name]
    test_clf.fit(X, y)

    test_suite = ClassificationTests(test_clf,
        test_data, target_name, column_names)
    classes = list(df.target.unique())
    lower_bound_requirement = {klass: 0.9 for klass in classes}
    assert test_suite.precision_lower_boundary_per_class(
        lower_bound_requirement
    )
```

Notice that we train on the production data and labels (in this case in target) and then test against the labels we know. Here we use the lower_bound_requirement variable to set the expectation for how well the model should do against the test set. If the labels generated by the production model train a model that performs as well on the test data as the production model did on the test set, then we have some confidence in the labels it produces. This is probably not the only way one could do this comparison, if you come up with something better, please share back out to the project!

CHAPTER 3

Designing your own tests

Before we jump into the API and all the premade tests that have been written to make your life easier, let's talk about a process for designing your own machine learning tests. The reason for doing this is important, machine learning testing is not like other software engineering tests. That's because software engineering tests are deterministic, like software engineering code ought to be. However, when you write tests for your data or your machine learning model, you need to account for the probabilistic nature of the code you are writing. The goal, therefore is much more fuzzy. But the process defined below should help you out.

3.1 It's About Proving Or Disproving Assumptions

There are a standard set of steps to any machine learning project:

1. Exploratory Analysis
2. Data Cleaning
3. Model Evaluation
4. Productionalizing The Model
5. Monitoring The Model

Machine learning tests are really about monitoring, but the big open question is, what do you monitor?

Monitoring the steps you took in 1-3 above, gives at least a base line. There will likely be other things to account for and monitor once you go into production, but what you've found in evaluation will likely be helpful later. So that should inform your first set of tests.

3.2 Data Monitoring Tests

Specifically, we can monitor the data by:

- checking to see if any descriptive statistics you found have changed substantially

- checking to see if current data is correlated with previous data per column
- checking to see if columns that were correlated or uncorrelated in past data remain that way
- checking to see if the number of clusters in the data has changed in a meaningful way
- checking to see whether the number of missing values stays consistent between new and old data,
- checking to see certain monotonicity requirements between columns remain consistent

It is an imperative to model the data because your model is merely a function of your data. If your data is bad or changes in some important way, your model will be useless. Also, there may be more measures you used to evaluate the data and those may become important features of whatever model you build later on. Therefore, making sure your data continues to follow the trends found previously may be of great import. Otherwise, your model might be wrong and you'd never know it.

3.3 Model Monitoring Tests

Additionally, we can monitor the model itself:

- checking to see if the model meets all metric requirements as specified by the business use-case
- checking to see if the model does better than some other test model on all measures of interest

3.4 System Monitoring Tests

Finally, there are also traditional tests one should run:

- making sure the serialized model exists where expected
- making sure the data exists where expected
- making sure data can flow into the system, to the model and through it
- making sure the new data matches the types you expect
- making sure the model produces the types you expect
- making sure new models can be deployed to the model pipeline
- making sure the model can perform well under load
- making sure the data can flow through fast enough to reach the model at ingress and egress

These three classes of machine learning system evaluation form a minimal reference set for monitoring such a system. There are likely more tests you'll need to write, but again just to outline the process in clear terms:

1. Look at what you wrote when you did exploratory analysis and data cleaning, turn those into tests to make sure your data stays that way, as long as it's supposed to
2. Look at how your model performed on test and training data, turn those evaluation measures into tests to make sure your model performs as well in production
3. Make sure everything actually goes from point A (the start of your system) to point B (the end of your system).

3.5 Fairness Monitoring Tests

There is a fourth class of tests that are unclear regarding the ethical nature of the algorithm you are building. These tests are unfortunately poorly defined at the present moment and very context specific, so all that can be offered is an example of what one might do:

Suppose you worked for a bank and were writing a piece of software that determined who gets a loan. Assuming a fair system folks from all races, genders, ages would get loans at a similar rate or would perhaps not be rejected due to race, gender, age or other factors.

If when accounting for some protected variable like race, gender, or age your algorithm does something odd compared to when not accounting for race, gender, or age then your algorithm may be biased.

However, this field of research is far from complete. There are some notions of testing for this, at the present moment they appear to be in need of further research and analysis. However, if possible, one should account for such a set of tests if possible, to ensure your algorithm is fair, unbiased and treats all individuals equally and fairly.

CHAPTER 4

Classification Tests

The goal of the following set of tests is to accomplish some monitoring goals:

1. Establish baselines for model performance in production per class
2. Establish maximum processing time for various volumes of data, through the statistical model
3. Ensure that the current model in production is the best available model according to a set of predefined measures

Let's look at each of these classes of tests now.

4.1 Lower Bound Classification Measures

Each of the following examples ensures that your classifier meets a minimum criteria, which should be decided based on the need of your use-case. One simple way to do this is to define failure by how many dollars it will cost you.

Precision, Recall and F1 score are great tools for ensuring your classifier optimizes for minimal misclassification, however you define it.

That is why they are basis of the set of tests found below.

4.2 Classifier Test Example - Model Metrics

Suppose you had the following model:

```
from sklearn import tree
import pandas as pd
import numpy as np
import joblib

df = pd.DataFrame()
for _ in range(1000):
    a = np.random.normal(0, 1)
```

(continues on next page)

(continued from previous page)

```
    b = np.random.normal(0, 3)
    c = np.random.normal(12, 4)
    if a + b + c > 11:
        target = 1
    else:
        target = 0
    df = df.append({
        "A": a,
        "B": b,
        "C": c,
        "target": target
    }, ignore_index=True)

clf = tree.DecisionTreeClassifier()
X = df[["A", "B", "C"]]
clf.fit(X, df["target"])
joblib.dump(clf, "model.joblib")
df.to_csv("data.csv")
```

We could write the following set of tests to ensure this model does well:

```
from drifter_ml.classification_tests import ClassificationTests
import joblib
import pandas as pd

def test_precision():
    df = pd.read_csv("data.csv")
    column_names = ["A", "B", "C"]
    target_name = "target"
    clf = joblib.load("model.joblib")

    test_suite = ClassificationTests(clf,
    df, target_name, column_names)
    classes = list(df.target.unique())
    assert test_suite.precision_lower_boundary_per_class(
        {klass: 0.9 for klass in classes}
    )

def test_recall():
    df = pd.read_csv("data.csv")
    column_names = ["A", "B", "C"]
    target_name = "target"
    clf = joblib.load("model.joblib")

    test_suite = ClassificationTests(clf,
    df, target_name, column_names)
    classes = list(df.target.unique())
    assert test_suite.recall_lower_boundary_per_class(
        {klass: 0.9 for klass in classes}
    )

def test_f1():
    df = pd.read_csv("data.csv")
    column_names = ["A", "B", "C"]
    target_name = "target"
    clf = joblib.load("model.joblib")
```

(continues on next page)

(continued from previous page)

```
    test_suite = ClassificationTests(clf,
    df, target_name, column_names)
    classes = list(df.target.unique())
    assert test_suite.f1_lower_boundary_per_class(
        {klass: 0.9 for klass in classes}
    )
```

Or you could simply write one test for all three:

```
from drifter_ml.classification_tests import ClassificationTests
import joblib
import pandas as pd

def test_precision_recall_f1():
    df = pd.read_csv("data.csv")
    column_names = ["A", "B", "C"]
    target_name = "target"
    clf = joblib.load("model.joblib")

    test_suite = ClassificationTests(clf,
    df, target_name, column_names)
    classes = list(df.target.unique())
    assert test_suite.classifier_testing_per_class(
        {klass: 0.9 for klass in classes},
        {klass: 0.9 for klass in classes},
        {klass: 0.9 for klass in classes}
    )
```

Regardless of which test you choose, you get complete flexibility to ensure your model always meets the minimum criteria so that your costs are minimized, given constraints.

4.3 Classifier Test Example - Model Speed

Additionally, you can test to ensure your classifier performs, even under load. Assume we have the same model as before:

```
from sklearn import tree
import pandas as pd
import numpy as np
import joblib

df = pd.DataFrame()
for _ in range(1000):
    a = np.random.normal(0, 1)
    b = np.random.normal(0, 3)
    c = np.random.normal(12, 4)
    if a + b + c > 11:
        target = 1
    else:
        target = 0
    df = df.append({
        "A": a,
        "B": b,
        "C": c,
```

(continues on next page)

(continued from previous page)

```
        "target": target
    }, ignore_index=True)

clf = tree.DecisionTreeClassifier()
X = df[["A", "B", "C"]]
clf.fit(X, df["target"])
joblib.dump(clf, "model.joblib")
df.to_csv("data.csv")
```

Now we test to ensure the model predicts new labels within our constraints:

```
from drifter_ml.classification_tests import ClassificationTests
import joblib
import pandas as pd

def test_precision_recall_f1_speed():
    df = pd.read_csv("data.csv")
    column_names = ["A", "B", "C"]
    target_name = "target"
    clf = joblib.load("model.joblib")

    test_suite = ClassificationTests(clf,
    df, target_name, column_names)
sample_sizes = [i for i in range(100, 1000, 100)]
max_run_times = [100 for _ in range(len(sample_sizes))]
    assert test_suite.run_time_stress_test(
        sample_sizes, max_run_times
    )
```

This test ensures that from 1 to 100000 elements, the model never takes longer than 10 seconds.

4.4 Cross Validation Based Testing

In the last section we asked questions of our model with respect to a lower boundary, both of various model measures as well as speed measurement in seconds. Now armed with cross validation we can ask questions about sections of our dataset, to ensure that the measures we found were an accurate representation across the dataset, rather than one global metric across the entire dataset. Just to make sure we are all on the same page, cross validation breaks the dataset into unique samples and then each sample is used as the test sample, all other samples are used as training, the score for each validation sample is recorded and then the model is discarded. For more information and a detailed introduction see https://machinelearningmastery.com/k-fold-cross-validation/.

The advantage of checking our model in this way is now it is less likely that the model is just memorizing the training data and will actually scale to other examples. This happens because the model scores are tested on a more limited dataset and also because "k" samples, the tuning parameter in cross validation, are tested to ensure the model performance is consistent.

This also yields some advantages for testing, because now we can verify that our lower boundary precision, recall or f1 score is true across many folds, rather than some global lower bound which may not be true on some subset of the data. This gives us more confidence in our models overall efficacy, but also requires that we have enough data to ensure our model can learn something.

Sadly I could find no good rules of thumb but I'd say less than you need at least something like 1000 data points per fold at least, and it's probably best to never go above 20 folds unless your dataset is truly massive, like in the gigabytes.

4.5 Classifier Test Example - Cross Validation Lower Bound Precision

This example won't be that different from what you've seen before, except now we can tune on the number of folds to include. Let's spice things up by using a keras classifier instead of a scikit learn one:

```
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
import pandas as pd
import numpy as np
import joblib

# Function to create model, required for KerasClassifier
def create_model():
    # create model
    model = Sequential()
    model.add(Dense(12, input_dim=3, activation='relu'))
    model.add(Dense(8, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model

# fix random seed for reproducibility
df = pd.DataFrame()
for _ in range(1000):
    a = np.random.normal(0, 1)
    b = np.random.normal(0, 3)
    c = np.random.normal(12, 4)
    if a + b + c > 11:
        target = 1
    else:
        target = 0
    df = df.append({
        "A": a,
        "B": b,
        "C": c,
        "target": target
    }, ignore_index=True)

# split into input (X) and output (Y) variables
# create model
clf = KerasClassifier(build_fn=create_model, epochs=150, batch_size=10, verbose=0)
X = df[["A", "B", "C"]]
clf.fit(X, df["target"])
joblib.dump(clf, "model.joblib")
df.to_csv("data.csv")
```

Now that we have the model and data saved, let's write the test:

```
from drifter_ml.classification_tests import ClassificationTests
import joblib
import pandas as pd

def test_cv_precision_lower_boundary():
```

(continues on next page)

(continued from previous page)

```
    df = pd.read_csv("data.csv")
    column_names = ["A", "B", "C"]
    target_name = "target"
    clf = joblib.load("model.joblib")

    test_suite = ClassificationTests(clf,
    df, target_name, column_names)
    lower_boundary = 0.9
    test_suite.cross_val_precision_lower_boundary(
        lower_boundary
    )
```

There are a few things to notice here:

1. The set up didn't change - we train the model the same way, we store the model the same way, we pass the model in the same way.
2. We aren't specifying percision per class - we will see examples of tests like that below, but because of the added stringency of limiting our training set, as well as training it across several samples of the dataset, sometimes called folds, we now don't need to specify as much granularity. What we are really testing here is somewhat different - we want to make sure no samples of the dataset form significantly worse than the average. What we are really looking for is anomalous samples of the data, that the model does much worse on. Because any training set is just a sample, if a given subsample does much worse than others, then we need to ask the question - is this given subsample representative of a pattern we may see in the future? Is it truly an anamoly? If it's not, that's usually a strong indicator that our model needs some work.

4.6 Classifier Test Example - Cross Validation Average

In the above example we test to ensure that none of the folds fall below a precision of 0.9 per fold. But what if we only care if one of the folds does significantly worse than the others? But don't actually care if all the folds meet the minimum criteria? After all, some level of any model measure is defined by how much data you train it on. It could be the case that we are right on the edge of having enough labeled data to train the model for all the imperative cases, but not enough to really ensure 90% percision, recall or some other meeasure. If that is the case, then we could simply look to see if any of the folds does significantly worse than some notion of centrality, which could be a red flag on its own.

Here we can set some deviance from the center for precision, recall or f1 score. If a given fold falls below some deviance from centrality then we believe some intervention needs to be taken. Let's look at an example:

```
from sklearn import tree
import pandas as pd
import numpy as np
import joblib

df = pd.DataFrame()
for _ in range(1000):
    a = np.random.normal(0, 1)
    b = np.random.normal(0, 3)
    c = np.random.normal(12, 4)
    if a + b + c > 11:
        target = 1
    else:
        target = 0
    df = df.append({
```

(continues on next page)

(continued from previous page)

```
        "A": a,
        "B": b,
        "C": c,
        "target": target
    }, ignore_index=True)

clf = tree.DecisionTreeClassifier()
X = df[["A", "B", "C"]]
clf.fit(X, df["target"])
joblib.dump(clf, "model.joblib")
df.to_csv("data.csv")
```

Let's see a test:

```
from drifter_ml.classification_tests import ClassificationTests
import joblib
import pandas as pd

def test_cv_precision_anomaly_detection():
    df = pd.read_csv("data.csv")
    column_names = ["A", "B", "C"]
    target_name = "target"
    clf = joblib.load("model.joblib")

    test_suite = ClassificationTests(clf,
    df, target_name, column_names)
    precision_tolerance = 0.2
    test_suite.cross_val_precision_anomaly_detection(
        precision_tolerance, method='mean'
    )
```

Here instead of setting an expectation of the precision, we set an expectation of the deviance from average precision. So if the average is 0.7 and one of the folds scores is less than 5.0 then the test fails. So it's important to have some lower boundary in place as well. However we can be less stringent if we include this test. A more complete test suite would likely be something like this:

```
from drifter_ml.classification_tests import ClassificationTests
import joblib
import pandas as pd

def test_cv_precision_anomaly_detection():
    df = pd.read_csv("data.csv")
    column_names = ["A", "B", "C"]
    target_name = "target"
    clf = joblib.load("model.joblib")

    test_suite = ClassificationTests(clf,
    df, target_name, column_names)
    precision_tolerance = 0.2
    test_suite.cross_val_precision_anomaly_detection(
        precision_tolerance, method='mean'
    )

def test_cv_precision_lower_boundary():
    df = pd.read_csv("data.csv")
    column_names = ["A", "B", "C"]
```

(continues on next page)

(continued from previous page)

```
    target_name = "target"
    clf = joblib.load("model.joblib")

    test_suite = ClassificationTests(clf,
    df, target_name, column_names)
    min_averange = 0.7
    test_suite.cross_val_precision_avg(
        min_average, method='mean'
    )
```

Now we can say for sure, the precision should be at least 0.7 on average but can fall below up to 0.2 of that before we raise an error. So

4.7 Classifier Test Example - Cross Validation Anamoly Detection With Spread

In the previous example, we looked for a specific deviance now we'll make use of some properties of statistics to define what exactly we mean by an anamolous fold. In order to do this, we'll look at deviance with respect to spread. To make this concrete, let's walk through what that means:

```
from drifter_ml.classification_tests import ClassificationTests
import joblib
import pandas as pd

def test_cv_precision_anomaly_detection():
    df = pd.read_csv("data.csv")
    column_names = ["A", "B", "C"]
    target_name = "target"
    clf = joblib.load("model.joblib")

    test_suite = ClassificationTests(clf,
    df, target_name, column_names)
    precision_tolerance = 0.2
    test_suite.cross_val_precision_anomaly_detection(
        precision_tolerance, method='mean'
    )
```

Before we go through what's happening let's recall what cross validation is. The basic notion of cross validation is random samples are taken, called folds of from the training set, trains the algorithm with that data and tests against all the other folds. For this reason, it is necessary that you have enough data such that you can learn a pattern from the data. For more information on k-fold check out this article: https://machinelearningmastery.com/k-fold-cross-validation/.

As you can see we require a precision tolerance of 0.2 per fold of the cross validation. To understand how this comes into play, let's look at how cross validation anomaly detection is done generally in the library:

1. decide on the measure of center to use
2. calculate the average of all the scores (each score comes from a fold)
3. compute the list of deviances from the average
4. determine if the deviance from the average is every greater than the tolerance

So basically, this is a test for consistency on different folds of the data. If the model performances above or below the tolerance bound on any of the folds, then the test fails. This is really good if you need your model to act in an expected way, a lot of the time.

CHAPTER 5

Regression Tests

So this section will likely be the most confusing for anyone coming from classical software engineering. Here regression refers to a model that outputs a floating point number, instead of a class. The biggest important difference between classification and regression is, the numbers produced by regression are "real" numbers. So they actually have magnitude, direction, a sense of scale, etc.

Classification returns a "class". Which means class "1" has no ordering relationship with class "2". So you shouldn't compare these with ordering.

In any event, the regression tests break out into the follow categories:

1. Establish a baseline maximum error tolerance based on a model measure
2. Establish a tolerance level for deviance from the average fold error
3. Stress testing for the speed of calculating new values
4. Comparison of the current model against new models for the above defined measures
5. Comparison of the speed of performance against new models

5.1 Upper Bound Regression Metrics

Each of the following examples ensures that your model meets a minimum criteria, which should be decided based on the need of your use-case. One simple way to do this is to define failure by how many dollars it will cost you for every unit amount your model is off on average.

Mean Squared Error and Median Absolute Error are great tools for ensuring your regressor optimizes for least error. The scale of that error will be entirely context specific.

That is why they are basis of the set of tests found below.

5.2 Regression Test Example - Model Metrics

Suppose you had the following model:

```
from sklearn import linear_model
import pandas as pd
import numpy as np
import joblib

df = pd.DataFrame()
for _ in range(1000):
    a = np.random.normal(0, 1)
    b = np.random.normal(0, 3)
    c = np.random.normal(12, 4)
    target = 5*a + 3*b + c
    df = df.append({
        "A": a,
        "B": b,
        "C": c,
        "target": target
    }, ignore_index=True)

reg = linear_model.LinearRegression()
X = df[["A", "B", "C"]]
reg.fit(X, df["target"])
joblib.dump(reg, "model.joblib")
df.to_csv("data.csv")
```

We could write the following set of tests to ensure this model does well:

```
from drifter_ml.regression_tests import RegressionTests
import joblib
import pandas as pd

def test_mse():
    df = pd.read_csv("data.csv")
    column_names = ["A", "B", "C"]
    target_name = "target"
    reg = joblib.load("model.joblib")

    test_suite = RegressionTests(reg,
    df, target_name, column_names)
    mse_boundary = 15
    assert test_suite.mse_upper_boundary(mse_boundary)

def test_mae():
    df = pd.read_csv("data.csv")
    column_names = ["A", "B", "C"]
    target_name = "target"
    reg = joblib.load("model.joblib")

    test_suite = RegressionTests(reg,
    df, target_name, column_names)
    mae_boundary = 10
    assert test_suite.mae_upper_boundary(mae_boundary)
```

Or you could simply write one test for all three:

```
from drifter_ml.regression_tests import RegressionTests
import joblib
import pandas as pd

def test_mse_mae():
    df = pd.read_csv("data.csv")
    column_names = ["A", "B", "C"]
    target_name = "target"
    reg = joblib.load("model.joblib")

    test_suite = RegressionTests(reg,
    df, target_name, column_names)
    mse_boundary = 15
    mae_boundary = 10
    assert test_suite.regression_testing(mse_boundary,
                                         mae_boundary)
```

5.3 Regression Test Example - Model Speed

Additionally, you can test to ensure your regressor performs, even under load. Assume we have the same model as before:

```
from sklearn import linear_model
import pandas as pd
import numpy as np
import joblib

df = pd.DataFrame()
for _ in range(1000):
    a = np.random.normal(0, 1)
    b = np.random.normal(0, 3)
    c = np.random.normal(12, 4)
    target = 5*a + 3*b + c
    df = df.append({
        "A": a,
        "B": b,
        "C": c,
        "target": target
    }, ignore_index=True)

reg = linear_model.LinearRegression()
X = df[["A", "B", "C"]]
reg.fit(X, df["target"])
joblib.dump(reg, "model.joblib")
df.to_csv("data.csv")
```

Now we test to ensure the model predicts new labels within our constraints:

```
from drifter_ml.regression_tests import RegressionTests
import joblib
import pandas as pd

def test_mse_mae_speed():
    df = pd.read_csv("data.csv")
    column_names = ["A", "B", "C"]
```

(continues on next page)

(continued from previous page)

```
    target_name = "target"
    reg = joblib.load("model.joblib")

    test_suite = RegressionTests(reg,
    df, target_name, column_names)
    performance_boundary = []
    for size in range(1, 100000, 100):
        performance_boundary.append({
                "sample_size": size,
                "max_run_time": 10.0 # seconds
        })
    assert test_suite.run_time_stress_test(
        performance_boundary
    )
```

This test ensures that from 1 to 100000 elements, the model never takes longer than 10 seconds.

CHAPTER 6

drifter_ml.classification_tests package

6.1 Submodules

6.2 drifter_ml.classification_tests.classification_tests module

class drifter_ml.classification_tests.classification_tests.**ClassificationTests**(*clf*, *test_data*, *target_name*, *column_names*)

Bases: *drifter_ml.classification_tests.classification_tests.FixedClassificationMetrics*

The general goal of this class it to test classification algorithms. The tests in this class move from simple to sophisticated:

- cross_val_average : the average of all folds must be above some number
- cross_val_lower_boundary : each fold must be above the lower boundary
- lower_boundary_per_class : each class must be above a given lower boundary the lower boundary per class can be different
- cross_val_anomaly_detection : the score for each fold must have a deviance from the average below a set tolerance
- cross_val_per_class_anomaly_detection : the score for each class for each fold must have a deviance from the average below a set tolerance

As you can see, at each level of sophistication we need more data to get representative sets. But if more data is available, then we are able to test increasingly more cases. The more data we have to test against, the more sure we can be about how well our model does.

Another lense to view each classes of tests, is with respect to stringency. If we need our model to absolutely work all the time, it might be important to use the most sophisticated class - something with cross validation,

per class. It's worth noting, that increased stringency isn't always a good thing. Statistical models, by definition aren't supposed to cover every case perfectly. They are supposed to be flexible. So you should only use the most strigent checks if you truly have a ton of data. Otherwise, you will more or less 'overfit' your test suite to try and look for errors. Testing in machine learning like in software engineering is very much an art. You need to be sure to cover enough cases, without going overboard.

classifier_testing_per_class(*precision_lower_boundary: dict*, *recall_lower_boundary: dict*, *f1_lower_boundary: dict*, *average='binary'*)

This is a slightly less naive stragey, it checks the: * precision score per class, * recall score per class, * f1 score per class Each class is boundary is mapped to the class via a dictionary allowing for different lower boundaries, per class. if any of the classes are less than the lower boundary, then False is returned.

Parameters

- **precision_lower_boundary** (*dict*) – the lower boundary for each class' precision score
- **recall_lower_boundary** (*dict*) – the lower boundary for each class' recall score
- **f1_lower_boundary** (*dict*) – the lower boundary for each class' f1 score
- **average** (*string*) – how to calculate the precision

Returns

- *True if all the classes of the precision scores are*
- *greater than the lower_boundary*
- *False if the classes for the precision scores are*
- *less than the lower_boundary*

cross_val_classifier_testing(*precision_lower_boundary: float*, *recall_lower_boundary: float*, *f1_lower_boundary: float*, *cv=3*, *average='binary'*)

runs the cross validated lower boundary methods for: * precision, * recall, * f1 score The basic idea for these three methods is to check if the accuracy metric stays above a given lower bound. We can set the same precision, recall, or f1 score lower boundary or specify each depending on necessary criteria.

Parameters

- **precision_lower_boundary** (*float*) – the lower boundary for a given precision score
- **recall_lower_boundary** (*float*) – the lower boundary for a given recall score
- **f1_lower_boundary** (*float*) – the lower boundary for a given f1 score
- **cv** (*int*) – the number of folds to consider
- **average** (*string*) – how to calculate the metrics (precision, recall, f1)

Returns

- *Returns True if precision, recall and f1 tests*
- *work.*
- *False otherwise*

cross_val_f1_anomaly_detection(*tolerance: float*, *cv=3*, *average='binary'*, *method='mean'*)

This checks the k fold (cross validation) f1 score, based on anolamies. The way the anomaly detection scheme works is, an average is calculated and then if the deviance from the average is greater than the set tolerance, then False is returned.

Parameters

- **tolerance** (*float*) – the tolerance from the average f1 score
- **cv** (*int*) – the number of folds to consider
- **average** (*string*) – how to calculate the f1 score
- **method** (*string*) – how to calculate the center

Returns

- *True if all the deviances from average for all the folds*
- *are above tolerance for f1 score*
- *False if any of the deviances from the average for any of*
- *the folds are below the tolerance for f1 score*

cross_val_f1_avg(*minimum_center_tolerance*, *cv=3*, *average='binary'*, *method='mean'*)

This generates the k fold (cross validation) f1 scores, then based on computes the average of those scores. The way the average scheme works is, an average is calculated and then if the average is less than the minimum tolerance, then False is returned.

Parameters

- **minimum_center_tolerance** (*float*) – the average f1 score must be greater than this number
- **cv** (*int*) – the number of folds to consider
- **average** (*string*) – how to calculate the f1 score
- **method** (*string*) – how to calculate the center

Returns

- *True if all the folds of the f1 score are greater than*
- *the minimum_center_tolerance*
- *False if the average folds for the f1 score are less than*
- *the minimum_center_tolerance*

cross_val_f1_lower_boundary(*lower_boundary*, *cv=3*, *average='binary'*)

This is possibly the most naive stragey, it generates the k fold (cross validation) f1 scores, if any of the k folds are less than the lower boundary, then False is returned.

Parameters

- **lower_boundary** (*float*) – the lower boundary for a given f1 score
- **cv** (*int*) – the number of folds to consider
- **average** (*string*) – how to calculate the f1 score

Returns

- *True if all the folds of the f1 scores are greater than*
- *the lower_boundary*
- *False if the folds for the f1 scores are less than*
- *the lower_boundary*

cross_val_per_class_f1_anomaly_detection(*tolerance: float, cv=3, average='binary', method='mean'*)

This checks the cross validated per class f1 score, based on anolamies. The way the anomaly detection scheme works is, an average is calculated and then if the deviance from the average is greater than the set tolerance, then False is returned.

Parameters

- **tolerance** (*float*) – the tolerance from the average f1 score
- **cv** (*int*) – the number of folds to consider
- **average** (*string*) – how to calculate the f1 score
- **method** (*string*) – how to calculate the center

Returns

- *True if all the deviances from average for all the folds*
- *are above tolerance for f1 score*
- *False if any of the deviances from the average for any of*
- *the folds are below the tolerance for f1 score*

cross_val_per_class_precision_anomaly_detection(*tolerance: float, cv=3, average='binary', method='mean'*)

This checks the cross validated per class percision score, based on anolamies. The way the anomaly detection scheme works is, an average is calculated and then if the deviance from the average is greater than the set tolerance, then False is returned.

Parameters

- **tolerance** (*float*) – the tolerance from the average precision
- **cv** (*int*) – the number of folds to consider
- **average** (*string*) – how to calculate the precision
- **method** (*string*) – how to calculate the center

Returns

- *True if all the deviances from average for all the folds*
- *are above tolerance for precision*
- *False if any of the deviances from the average for any of*
- *the folds are below the tolerance for precision*

cross_val_per_class_recall_anomaly_detection(*tolerance: float, cv=3, average='binary', method='mean'*)

This checks the cross validated per class recall score, based on anolamies. The way the anomaly detection scheme works is, an average is calculated and then if the deviance from the average is greater than the set tolerance, then False is returned.

Parameters

- **tolerance** (*float*) – the tolerance from the average recall
- **cv** (*int*) – the number of folds to consider
- **average** (*string*) – how to calculate the recall
- **method** (*string*) – how to calculate the center

Returns

- *True if all the deviances from average for all the folds*
- *are above tolerance for recall*
- *False if any of the deviances from the average for any of*
- *the folds are below the tolerance for recall*

cross_val_per_class_roc_auc_anomaly_detection(*tolerance: float, cv=3, average='micro', method='mean'*)

This checks the cross validated per class roc auc score, based on anolamies. The way the anomaly detection scheme works is, an average is calculated and then if the deviance from the average is greater than the set tolerance, then False is returned.

Parameters

- **tolerance** (*float*) – the tolerance from the average roc auc
- **cv** (*int*) – the number of folds to consider
- **average** (*string*) – how to calculate the roc auc
- **method** (*string*) – how to calculate the center

Returns

- *True if all the deviances from average for all the folds*
- *are above tolerance for roc auc*
- *False if any of the deviances from the average for any of*
- *the folds are below the tolerance for roc auc*

cross_val_precision_anomaly_detection(*tolerance: float, cv=3, average='binary', method='mean'*)

This checks the k fold (cross validation) precision score, based on anolamies. The way the anomaly detection scheme works is, an average is calculated and then if the deviance from the average is greater than the set tolerance, then False is returned.

Parameters

- **tolerance** (*float*) – the tolerance from the average precision
- **cv** (*int*) – the number of folds to consider
- **average** (*string*) – how to calculate the precision
- **method** (*string*) – how to calculate the center

Returns

- *True if all the deviances from average for all the folds*
- *are above tolerance for precision*
- *False if any of the deviances from the average for any of*
- *the folds are below the tolerance for precision*

cross_val_precision_avg(*minimum_center_tolerance, cv=3, average='binary', method='mean'*)

This generates the k fold (cross validation) precision scores, then based on computes the average of those scores. The way the average scheme works is, an average is calculated and then if the average is less than the minimum tolerance, then False is returned.

Parameters

- **minimum_center_tolerance** (*float*) – the average precision must be greater than this number
- **cv** (*int*) – the number of folds to consider
- **average** (*string*) – how to calculate the precision
- **method** (*string*) – how to calculate the center

Returns

- *True if all the folds of the precision are greater than*
- *the minimum_center_tolerance*
- *False if the average folds for the precision are less than*
- *the minimum_center_tolerance*

cross_val_precision_lower_boundary(*lower_boundary*, *cv=3*, *average='binary'*)

This is possibly the most naive stragey, it generates the k fold (cross validation) precision scores, if any of the k folds are less than the lower boundary, then False is returned.

Parameters

- **lower_boundary** (*float*) – the lower boundary for a given precision score
- **cv** (*int*) – the number of folds to consider
- **average** (*string*) – how to calculate the precision

Returns

- *True if all the folds of the precision scores are*
- *greater than the lower_boundary*
- *False if the folds for the precision scores are*
- *less than the lower_boundary*

cross_val_recall_anomaly_detection(*tolerance: float*, *cv=3*, *average='binary'*, *method='mean'*)

This checks the k fold (cross validation) recall score, based on anolamies. The way the anomaly detection scheme works is, an average is calculated and then if the deviance from the average is greater than the set tolerance, then False is returned.

Parameters

- **tolerance** (*float*) – the tolerance from the average recall
- **cv** (*int*) – the number of folds to consider
- **average** (*string*) – how to calculate the recall
- **method** (*string*) – how to calculate the center

Returns

- *True if all the deviances from average for all the folds*
- *are above tolerance for recall*
- *False if any of the deviances from the average for any of*
- *the folds are below the tolerance for recall*

cross_val_recall_avg(*minimum_center_tolerance*, *cv=3*, *average='binary'*, *method='mean'*)
This generates the k fold (cross validation) recall scores, then based on computes the average of those scores. The way the average scheme works is, an average is calculated and then if the average is less than the minimum tolerance, then False is returned.

Parameters

- **minimum_center_tolerance** (*float*) – the average recall must be greater than this number
- **cv** (*int*) – the number of folds to consider
- **average** (*string*) – how to calculate the recall
- **method** (*string*) – how to calculate the center

Returns

- *True if all the folds of the recall are greater than*
- *the minimum_center_tolerance*
- *False if the average folds for the recall are less than*
- *the minimum_center_tolerance*

cross_val_recall_lower_boundary(*lower_boundary*, *cv=3*, *average='binary'*)
This is possibly the most naive stragey, it generates the k fold (cross validation) recall scores, if any of the k folds are less than the lower boundary, then False is returned.

Parameters

- **lower_boundary** (*float*) – the lower boundary for a given recall score
- **cv** (*int*) – the number of folds to consider
- **average** (*string*) – how to calculate the recall

Returns

- *True if all the folds of the recall scores are greater than*
- *the lower_boundary*
- *False if the folds for the recall scores are less than*
- *the lower_boundary*

cross_val_roc_auc_anomaly_detection(*tolerance: float*, *cv=3*, *average='micro'*, *method='mean'*)
This checks the k fold (cross validation) roc auc score, based on anolamies. The way the anomaly detection scheme works is, an average is calculated and then if the deviance from the average is greater than the set tolerance, then False is returned.

Parameters

- **tolerance** (*float*) – the tolerance from the average roc auc
- **cv** (*int*) – the number of folds to consider
- **average** (*string*) – how to calculate the roc auc
- **method** (*string*) – how to calculate the center

Returns

- *True if all the deviances from average for all the folds*
- *are above tolerance for roc auc*

- *False if any of the deviances from the average for any of*
- *the folds are below the tolerance for roc auc*

cross_val_roc_auc_avg(*minimum_center_tolerance*, *cv=3*, *average='micro'*, *method='mean'*)
This generates the k fold (cross validation) roc auc scores, then based on computes the average of those scores. The way the average scheme works is, an average is calculated and then if the average is less than the minimum tolerance, then False is returned.

Parameters

- **minimum_center_tolerance** (*float*) – the average roc auc must be greater than this number
- **cv** (*int*) – the number of folds to consider
- **average** (*string*) – how to calculate the roc auc
- **method** (*string*) – how to calculate the center

Returns

- *True if all the folds of the roc auc are greater than*
- *the minimum_center_tolerance*
- *False if the average folds for the roc auc are less than*
- *the minimum_center_tolerance*

cross_val_roc_auc_lower_boundary(*lower_boundary*, *cv=3*, *average='micro'*)
This is possibly the most naive stragey, it generates the k fold (cross validation) roc auc scores, if any of the k folds are less than the lower boundary, then False is returned.

Parameters

- **lower_boundary** (*float*) – the lower boundary for a given roc auc score
- **cv** (*int*) – the number of folds to consider
- **average** (*string*) – how to calculate the roc auc

Returns

- *True if all the folds of the roc auc scores are greater than*
- *the lower_boundary*
- *False if the folds for the roc auc scores are less than*
- *the lower_boundary*

describe_scores(*scores*, *method*)
Describes scores.

Parameters

- **scores** (*array-like*) – the scores from the model, as a list or numpy array
- **method** (*string*) – the method to use to calculate central tendency and spread

Returns

- *Returns the central tendency, and spread*
- *by method.*
- *Methods*

- *mean*
- * **central tendency** (*mean*)
- * **spread** (*standard deviation*)
- *median*
- * **central tendency** (*median*)
- * **spread** (*interquartile range*)
- *trimean*
- * **central tendency** (*trimean*)
- * **spread** (*trimean absolute deviation*)

f1_cv (*cv*, *average='binary'*)
This method performs cross-validation over f1-score.

Parameters

- **cv** ($\star$) – The number of cross validation folds to perform
- **average** ($\star$) – [None, 'binary'(default), 'micro', 'macro', 'samples', 'weighted'] This parameter is required for multiclass/multilabel targets. If None, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data.

  'binary': Only report results for the class specified by pos_label. This is applicable only if targets (y_{true, pred}) are binary.

  'micro': Calculate metrics globally by counting the total true positives, false negatives and false positives.

  'macro':

  Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

  'weighted': Calculate metrics for each label, and find their average weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance; it can result in an F-score that isnot between precision and recall.

  'samples': Calculate metrics for each instance, and find their average (only meaningful for multilabel classification where this differs from accuracy_score).

Returns

Return type Returns a scores of the k-fold f1-score.

f1_lower_boundary_per_class (*lower_boundary: dict*, *average='binary'*)
This is a slightly less naive stragey, it checks the f1 score, Each class is boundary is mapped to the class via a dictionary allowing for different lower boundaries, per class. if any of the classes are less than the lower boundary, then False is returned.

Parameters

- **lower_boundary** (*dict*) – the lower boundary for each class' f1 score
- **average** (*string*) – how to calculate the f1

Returns

- *True if all the classes of the f1 scores are*

- *greater than the lower_boundary*
- *False if the classes for the f1 scores are*
- *less than the lower_boundary*

get_test_score(*cross_val_dict*)

is_binary()
If number of classes == 2 returns True False otherwise

precision_cv(*cv*, *average='binary'*)
This method performs cross-validation over precision.

Parameters

- **cv** (*) – The number of cross validation folds to perform
- **average** (*) – [None, 'binary'(default), 'micro', 'macro', 'samples', 'weighted'] This parameter is required for multiclass/multilabel targets. If None, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data.

  'binary': Only report results for the class specified by pos_label. This is applicable only if targets (y_{true, pred}) are binary.

  'micro': Calculate metrics globally by counting the total true positives, false negatives and false positives.

  'macro':

  Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

  'weighted': Calculate metrics for each label, and find their average weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance; it can result in an F-score that isnot between precision and recall.

  'samples': Calculate metrics for each instance, and find their average (only meaningful for multilabel classification where this differs from accuracy_score).

Returns

Return type Returns a scores of the k-fold precision.

precision_lower_boundary_per_class(*lower_boundary: dict*, *average='binary'*)
This is a slightly less naive stragey, it checks the precision score, Each class is boundary is mapped to the class via a dictionary allowing for different lower boundaries, per class. if any of the classes are less than the lower boundary, then False is returned.

Parameters

- **lower_boundary** (*dict*) – the lower boundary for each class' precision score
- **average** (*string*) – how to calculate the precision

Returns

- *True if all the classes of the precision scores are*
- *greater than the lower_boundary*
- *False if the classes for the precision scores are*

- *less than the lower_boundary*

recall_cv(*cv*, *average='binary'*)

This method performs cross-validation over recall.

Parameters

- **cv** (*) – The number of cross validation folds to perform
- **average** (*) – [None, 'binary'(default), 'micro', 'macro', 'samples', 'weighted'] This parameter is required for multiclass/multilabel targets. If None, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data.

  'binary': Only report results for the class specified by pos_label. This is applicable only if targets (y_{true, pred}) are binary.

  'micro': Calculate metrics globally by counting the total true positives, false negatives and false positives.

  'macro':

  Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

  'weighted': Calculate metrics for each label, and find their average weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance; it can result in an F-score that isnot between precision and recall.

  'samples': Calculate metrics for each instance, and find their average (only meaningful for multilabel classification where this differs from accuracy_score).

Returns

Return type Returns a scores of the k-fold recall.

recall_lower_boundary_per_class(*lower_boundary: dict*, *average='binary'*)

This is a slightly less naive stragey, it checks the recall score, Each class is boundary is mapped to the class via a dictionary allowing for different lower boundaries, per class. if any of the classes are less than the lower boundary, then False is returned.

Parameters

- **lower_boundary** (*dict*) – the lower boundary for each class' recall score
- **average** (*string*) – how to calculate the recall

Returns

- *True if all the classes of the recall scores are*
- *greater than the lower_boundary*
- *False if the classes for the recall scores are*
- *less than the lower_boundary*

reset_average(*average*)

Resets the average to the correct thing. If the number of classes are not binary, Then average is changed to micro. Otherwise, return the current average.

roc_auc_cv(*cv*, *average='micro'*)

This method performs cross-validation over roc_auc.

Parameters

- **cv** (*) – The number of cross validation folds to perform
- **average** (*) – [None, 'binary'(default), 'micro', 'macro', 'samples', 'weighted'] This parameter is required for multiclass/multilabel targets. If None, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data.

  'binary': Only report results for the class specified by pos_label. This is applicable only if targets (y_{true, pred}) are binary.

  'micro': Calculate metrics globally by counting the total true positives, false negatives and false positives.

  'macro':

  Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

  'weighted': Calculate metrics for each label, and find their average weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance; it can result in an F-score that isnot between precision and recall.

  'samples': Calculate metrics for each instance, and find their average (only meaningful for multilabel classification where this differs from accuracy_score).

Returns

Return type Returns a scores of the k-fold roc_auc.

roc_auc_exception()
Ensures roc_auc score is used correctly. ROC AUC is only defined for binary classification.

roc_auc_lower_boundary_per_class(*lower_boundary: dict*, *average='micro'*)
This is a slightly less naive stragey, it checks the roc auc score, Each class is boundary is mapped to the class via a dictionary allowing for different lower boundaries, per class. if any of the classes are less than the lower boundary, then False is returned.

Parameters

- **lower_boundary** (*dict*) – the lower boundary for each class' roc auc score
- **average** (*string*) – how to calculate the roc auc

Returns

- *True if all the classes of the roc auc scores are*
- *greater than the lower_boundary*
- *False if the classes for the roc auc scores are*
- *less than the lower_boundary*

run_energy_stress_test(*sample_sizes: list*, *max_energy_usages: list*, *print_to_screen=False*, *print_to_pdf=False*)
This is a performance test to ensure that the model is energy efficient.

Note: the model must take longer than 5 seconds to run otherwise energyusage cannot accurate estimate the energy cost. At this point, the cost is neglible. Therefore, when testing, please try to use reasonable size estimates based on expected throughput.

Parameters

- **sample_sizes** (*list*) – the size of each sample to test for doing a prediction, each sample size is an integer
- **max_energy_usages** (*list*) – the maximum time in seconds that each sample should take to predict, at a maximum.

Returns

- *True if all samples predict within the maximum allowed*
- *energy usage.*
- *False otherwise.*

run_time_stress_test(*sample_sizes: list*, *max_run_times: list*)

This is a performance test to ensure that the model runs fast enough.

sample_sizes [list] the size of each sample to test for doing a prediction, each sample size is an integer

max_run_times [list] the maximum time in seconds that each sample should take to predict, at a maximum.

Returns

- *True if all samples predict within the maximum allowed*
- *time.*
- *False otherwise.*

spread_cross_val_classifier_testing(*precision_tolerance: float*, *recall_tolerance: float*, *f1_tolerance: float*, *method='mean'*, *cv=10*, *average='binary'*)

This is a somewhat intelligent stragey, it generates the k fold (cross validation) the following scores: * precision scores, * recall scores * f1 scores if any of the k folds score less than the center - (spread * tolerance), then False is returned.

Parameters

- **tolerance** (*float*) – the tolerance modifier for how far below the center the score can be before a false is returned
- **method** (*string*) – see describe for more details. * mean : the center is the mean, the spread is standard

  deviation.

  - **median** [the center is the median, the spread is] the interquartile range.
  - **trimean** [the center is the trimean, the spread is] trimean absolute deviation.

- **average** (*string*) – how to calculate the precision

Returns

- *True if all the folds of the precision, recall, f1 scores*
- *are greater than the center - (spread * tolerance)*
- *False if the folds for the precision, recall, f1 scores*
- *are less than the center - (spread * tolerance)*

spread_cross_val_f1_anomaly_detection(*tolerance*, *method='mean'*, *cv=10*, *average='binary'*)

This is a somewhat intelligent stragey, it generates the k fold (cross validation) f1 scores, if any of the k folds score less than the center - (spread * tolerance), then False is returned.

Parameters

- **tolerance** (*float*) – the tolerance modifier for how far below the center the score can be before a false is returned
- **method** (*string*) – see describe for more details. * mean : the center is the mean, the spread is standard

  deviation.

  - **median** [the center is the median, the spread is] the interquartile range.
  - **trimean** [the center is the trimean, the spread is] trimean absolute deviation.
- **average** (*string*) – how to calculate the f1 score

Returns

- *True if all the folds of the f1 scores are greater than*
- *the center - (spread * tolerance)*
- *False if the folds for the f1 scores are less than*
- *the center - (spread * tolerance)*

spread_cross_val_precision_anomaly_detection(*tolerance*, *method='mean'*, *cv=10*, *average='binary'*)

This is a somewhat intelligent stragey, it generates the k fold (cross validation) precision scores, if any of the k folds score less than the center - (spread * tolerance), then False is returned.

Parameters

- **tolerance** (*float*) – the tolerance modifier for how far below the center the score can be before a false is returned
- **method** (*string*) – see describe for more details. * mean : the center is the mean, the spread is standard

  deviation.

  - **median** [the center is the median, the spread is] the interquartile range.
  - **trimean** [the center is the trimean, the spread is] trimean absolute deviation.
- **average** (*string*) – how to calculate the precision

Returns

- *True if all the folds of the precision scores are greater than*
- *the center - (spread * tolerance)*
- *False if the folds for the precision scores are less than*
- *the center - (spread * tolerance)*

spread_cross_val_recall_anomaly_detection(*tolerance*, *method='mean'*, *cv=3*, *average='binary'*)

This is a somewhat intelligent stragey, it generates the k fold (cross validation) recall scores, if any of the k folds score less than the center - (spread * tolerance), then False is returned.

Parameters

- **tolerance** (*float*) – the tolerance modifier for how far below the center the score can be before a false is returned
- **method** (*string*) – see describe for more details. * mean : the center is the mean, the spread is standard

  deviation.

  - **median** [the center is the median, the spread is] the interquartile range.
  - **trimean** [the center is the trimean, the spread is] trimean absolute deviation.
- **average** (*string*) – how to calculate the recall

Returns

- *True if all the folds of the recall scores are greater than*
- *the center - (spread * tolerance)*
- *False if the folds for the recall scores are less than*
- *the center - (spread * tolerance)*

spread_cross_val_roc_auc_anomaly_detection(*tolerance*, *method='mean'*, *cv=10*, *average='micro'*)

This is a somewhat intelligent stragey, it generates the k fold (cross validation) roc auc scores, if any of the k folds score less than the center - (spread * tolerance), then False is returned.

Parameters

- **tolerance** (*float*) – the tolerance modifier for how far below the center the score can be before a false is returned
- **method** (*string*) – see describe for more details. * mean : the center is the mean, the spread is standard

  deviation.

  - **median** [the center is the median, the spread is] the interquartile range.
  - **trimean** [the center is the trimean, the spread is] trimean absolute deviation.
- **average** (*string*) – how to calculate the precision

Returns

- *True if all the folds of the roc auc scores are greater than*
- *the center - (spread * tolerance)*
- *False if the folds for the roc auc scores are less than*
- *the center - (spread * tolerance)*

trimean(*data*)

I'm exposing this as a public method because the trimean is not implemented in enough packages.

Formula: (25th percentile + 2*50th percentile + 75th percentile)/4

Parameters **data** (*array-like*) – an iterable, either a list or a numpy array

Returns the trimean

Return type float

trimean_absolute_deviation(*data*)
The trimean absolute deviation is the the average distance from the trimean.

Parameters **data** (*array-like*) – an iterable, either a list or a numpy array

Returns the average distance to the trimean

Return type float

class drifter_ml.classification_tests.classification_tests.**ClassifierComparison**(*clf_one*, *clf_two*, *test_data*, *target_name*, *column_names*)

Bases: *drifter_ml.classification_tests.classification_tests.FixedClassificationMetrics*

cross_val_f1(*clf*, *cv=3*, *average='binary'*)

cross_val_f1_per_class(*clf*, *cv=3*, *average='binary'*)

cross_val_per_class_two_model_classifier_testing(*cv=3*, *average='binary'*)

cross_val_precision(*clf*, *cv=3*, *average='binary'*)

cross_val_precision_per_class(*clf*, *cv=3*, *average='binary'*)

cross_val_recall(*clf*, *cv=3*, *average='binary'*)

cross_val_recall_per_class(*clf*, *cv=3*, *average='binary'*)

cross_val_roc_auc(*clf*, *cv=3*, *average='micro'*)

cross_val_roc_auc_per_class(*clf*, *cv=3*, *average='micro'*)

cross_val_two_model_classifier_testing(*cv=3*, *average='binary'*)

f1_per_class(*clf*, *average='binary'*)

is_binary()

precision_per_class(*clf*, *average='binary'*)

recall_per_class(*clf*, *average='binary'*)

reset_average(*average*)

roc_auc_exception()

roc_auc_per_class(*clf*, *average='micro'*)

two_model_classifier_testing(*average='binary'*)

two_model_prediction_run_time_stress_test(*sample_sizes*)

class drifter_ml.classification_tests.classification_tests.**FixedClassificationMetrics**
Bases: object

f1_score(*y_true*, *y_pred*, *labels=None*, *pos_label=1*, *average='binary'*, *sample_weight=None*)

The Scikit-Learn precision score, see the full documentation here: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_score.html

The difference between this f1 score and the one in scikit-learn, is we fix a small bug. When all the values in y_true are zero and y_pred are zero the f1_score returns one. (Which Scikit-learn does not do at present).

Parameters

- **y_true** (*1d array-like, or label indicator array / sparse matrix*) – Ground truth (correct) target values.
- **y_pred** (*1d array-like, or label indicator array / sparse matrix*) – Estimated targets as returned by a classifier
- **labels** (*list, optional*) – The set of labels to include when average != binary, and their order if average is None. Labels present in the data can be excluded, for example to calculate a multiclass average ignoring a majority negative class, while labels not present in the data will result in 0 components in a macro average. For multilabel targets, labels are column indices. By default, all labels in y_true and y_pred are used in sorted order.
- **pos_label** (*) – The class to report if average='binary' and the data is binary. If the data are multiclass or multilabel, this will be ignored; setting labels=[pos_label] and average != 'binary' will report scores for that label only.
- **average** (*) – [None, 'binary'(default), 'micro', 'macro', 'samples', 'weighted'] This parameter is required for multiclass/multilabel targets. If None, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data.

  'binary': Only report results for the class specified by pos_label. This is applicable only if targets (y_{true, pred}) are binary.

  'micro': Calculate metrics globally by counting the total true positives, false negatives and false positives.

  'macro':

  Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

  'weighted': Calculate metrics for each label, and find their average weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance; it can result in an F-score that isnot between precision and recall.

  'samples': Calculate metrics for each instance, and find their average (only meaningful for multilabel classification where this differs from accuracy_score).

- **sample_weight** (*) – array-like of shape = [n_samples], optional. Sample weights.

Returns *** f1** – (if average is not None) or array of float, shape = [n_unique_labels] F1 score of the positive class in binary classification or weighted average of the f1 scores of each class for the multiclass task.

Return type float

precision_score(*y_true*, *y_pred*, *labels=None*, *pos_label=1*, *average='binary'*, *sample_weight=None*)

The Scikit-Learn precision score, see the full documentation here: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_score.html

The difference between this precision score and the one in scikit-learn, is we fix a small bug. When all the values in y_true are zero and y_pred are zero the precision_score returns one. (Which Scikit-learn does not do at present).

Parameters

- **y_true** (*1d array-like, or label indicator array / sparse matrix*) – Ground truth (correct) target values.
- **y_pred** (*1d array-like, or label indicator array / sparse matrix*) – Estimated targets as returned by a classifier
- **labels** (*list, optional*) – The set of labels to include when average != binary, and their order if average is None. Labels present in the data can be excluded, for example to calculate a multiclass average ignoring a majority negative class, while labels not present in the data will result in 0 components in a macro average. For multilabel targets, labels are column indices. By default, all labels in y_true and y_pred are used in sorted order.
- **pos_label** (*str or int, 1 by default*) – The class to report if average='binary' and the data is binary. If the data are multiclass or multilabel, this will be ignored; setting labels=[pos_label] and average != 'binary' will report scores for that label only.
- **average** (*string,*) – [None, 'binary'(default), 'micro', 'macro', 'samples', 'weighted'] This parameter is required for multiclass/multilabel targets. If None, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data.

  'binary' [string] Only report results for the class specified by pos_label. This is applicable only if targets (y_{true, pred}) are binary.

  'micro' [string] Calculate metrics globally by counting the total true positives, false negatives and false positives.

  'macro' [string]

  Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

  'weighted' [string] Calculate metrics for each label, and find their average weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance; it can result in an F-score that isnot between precision and recall.

  'samples' [string] Calculate metrics for each instance, and find their average (only meaningful for multilabel classification where this differs from accuracy_score).

- **sample_weight** (*array-like*) –

  array-like of shape = [n_samples], optional Sample weights.

Returns **precision** – (if average is not None) or array of float, shape = [n_unique_labels] Precision of the positive class in binary classification or weighted average of the precision of each class for the multiclass task.

Return type float

recall_score(*y_true*, *y_pred*, *labels=None*, *pos_label=1*, *average='binary'*, *sample_weight=None*)

The Scikit-Learn precision score, see the full documentation here: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_score.html

The difference between this recall score and the one in scikit-learn, is we fix a small bug. When all the values in y_true are zero and y_pred are zero the recall_score returns one. (Which Scikit-learn does not do at present).

Parameters

- **y_true** (*1d array-like, or label indicator array / sparse matrix*) – Ground truth (correct) target values.
- **y_pred** (*1d array-like, or label indicator array / sparse matrix*) – Estimated targets as returned by a classifier
- **labels** (*list, optional*) – The set of labels to include when average != binary, and their order if average is None. Labels present in the data can be excluded, for example to calculate a multiclass average ignoring a majority negative class, while labels not present in the data will result in 0 components in a macro average. For multilabel targets, labels are column indices. By default, all labels in y_true and y_pred are used in sorted order.
- **pos_label** (*str or int, 1 by default*) – The class to report if average='binary' and the data is binary. If the data are multiclass or multilabel, this will be ignored; setting labels=[pos_label] and average != 'binary' will report scores for that label only.
- **average** (*string,*) – [None, 'binary'(default), 'micro', 'macro', 'samples', 'weighted'] This parameter is required for multiclass/multilabel targets. If None, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data.

  'binary': Only report results for the class specified by pos_label. This is applicable only if targets (y_{true, pred}) are binary.

  'micro': Calculate metrics globally by counting the total true positives, false negatives and false positives.

  'macro':

  Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

  'weighted': Calculate metrics for each label, and find their average weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance; it can result in an F-score that isnot between precision and recall.

  'samples': Calculate metrics for each instance, and find their average (only meaningful for multilabel classification where this differs from accuracy_score).

- **sample_weight** (*array-like*) – array-like of shape = [n_samples], optional Sample weights.

Returns **recall** – (if average is not None) or array of float, shape = [n_unique_labels] Recall of the positive class in binary classification or weighted average of the recall of each class for the multiclass task.

Return type float

roc_auc_score(*y_true*, *y_pred*, *labels=None*, *pos_label=1*, *average='micro'*, *sample_weight=None*)

The Scikit-Learn precision score, see the full documentation here: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_score.html

The difference between this roc_auc score and the one in scikit-learn, is we fix a small bug. When all the values in y_true are zero and y_pred are zero the roc_auc_score returns one. (Which Scikit-learn does not do at present).

Parameters

- **y_true** (*) – Ground truth (correct) target values.
- **y_pred** (*) – Estimated targets as returned by a classifier
- **labels** (*) – The set of labels to include when average != binary, and their order if average is None. Labels present in the data can be excluded, for example to calculate a multiclass average ignoring a majority negative class, while labels not present in the data will result in 0 components in a macro average. For multilabel targets, labels are column indices. By default, all labels in y_true and y_pred are used in sorted order.
- **pos_label** (*) – The class to report if average='binary' and the data is binary. If the data are multiclass or multilabel, this will be ignored; setting labels=[pos_label] and average != 'binary' will report scores for that label only.
- **average** (*) – string, [None, 'binary'(default), 'micro', 'macro', 'samples', 'weighted'] This parameter is required for multiclass/multilabel targets. If None, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data.

  'binary': Only report results for the class specified by pos_label. This is applicable only if targets (y_{true, pred}) are binary.

  'micro': Calculate metrics globally by counting the total true positives, false negatives and false positives.

  'macro':

  Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

  'weighted': Calculate metrics for each label, and find their average weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance; it can result in an F-score that isnot between precision and recall.

  'samples': Calculate metrics for each instance, and find their average (only meaningful for multilabel classification where this differs from accuracy_score).

- **sample_weight** (*) – array-like of shape = [n_samples], optional Sample weights.

Returns * **roc_auc** – (if average is not None) or array of float, shape = [n_unique_labels] Roc_auc score of the positive class in binary classification or weighted average of the roc_auc scores of each class for the multiclass task.

Return type float

6.3 Module contents

class drifter_ml.classification_tests.**ClassificationTests**(*clf*, *test_data*, *target_name*, *column_names*)

Bases: *drifter_ml.classification_tests.classification_tests.FixedClassificationMetrics*

The general goal of this class it to test classification algorithms. The tests in this class move from simple to sophisticated:

- cross_val_average : the average of all folds must be above some number
- cross_val_lower_boundary : each fold must be above the lower boundary
- lower_boundary_per_class : each class must be above a given lower boundary the lower boundary per class can be different
- cross_val_anomaly_detection : the score for each fold must have a deviance from the average below a set tolerance
- cross_val_per_class_anomaly_detection : the score for each class for each fold must have a deviance from the average below a set tolerance

As you can see, at each level of sophistication we need more data to get representative sets. But if more data is available, then we are able to test increasingly more cases. The more data we have to test against, the more sure we can be about how well our model does.

Another lense to view each classes of tests, is with respect to stringency. If we need our model to absolutely work all the time, it might be important to use the most sophisticated class - something with cross validation, per class. It's worth noting, that increased stringency isn't always a good thing. Statistical models, by definition aren't supposed to cover every case perfectly. They are supposed to be flexible. So you should only use the most strigent checks if you truly have a ton of data. Otherwise, you will more or less 'overfit' your test suite to try and look for errors. Testing in machine learning like in software engineering is very much an art. You need to be sure to cover enough cases, without going overboard.

classifier_testing_per_class(*precision_lower_boundary: dict*, *recall_lower_boundary: dict*, *f1_lower_boundary: dict*, *average='binary'*)

This is a slightly less naive stragey, it checks the: * precision score per class, * recall score per class, * f1 score per class Each class is boundary is mapped to the class via a dictionary allowing for different lower boundaries, per class. if any of the classes are less than the lower boundary, then False is returned.

Parameters

- **precision_lower_boundary** (*dict*) – the lower boundary for each class' precision score
- **recall_lower_boundary** (*dict*) – the lower boundary for each class' recall score
- **f1_lower_boundary** (*dict*) – the lower boundary for each class' f1 score
- **average** (*string*) – how to calculate the precision

Returns

- *True if all the classes of the precision scores are*
- *greater than the lower_boundary*
- *False if the classes for the precision scores are*
- *less than the lower_boundary*

cross_val_classifier_testing(*precision_lower_boundary: float*, *recall_lower_boundary: float*, *f1_lower_boundary: float*, *cv=3*, *average='binary'*)

runs the cross validated lower boundary methods for: * precision, * recall, * f1 score The basic idea for these three methods is to check if the accuracy metric stays above a given lower bound. We can set the same precision, recall, or f1 score lower boundary or specify each depending on necessary criteria.

Parameters

- **precision_lower_boundary** (*float*) – the lower boundary for a given precision score
- **recall_lower_boundary** (*float*) – the lower boundary for a given recall score
- **f1_lower_boundary** (*float*) – the lower boundary for a given f1 score
- **cv** (*int*) – the number of folds to consider
- **average** (*string*) – how to calculate the metrics (precision, recall, f1)

Returns

- *Returns True if precision, recall and f1 tests*
- *work.*
- *False otherwise*

cross_val_f1_anomaly_detection(*tolerance: float*, *cv=3*, *average='binary'*, *method='mean'*)

This checks the k fold (cross validation) f1 score, based on anolamies. The way the anomaly detection scheme works is, an average is calculated and then if the deviance from the average is greater than the set tolerance, then False is returned.

Parameters

- **tolerance** (*float*) – the tolerance from the average f1 score
- **cv** (*int*) – the number of folds to consider
- **average** (*string*) – how to calculate the f1 score
- **method** (*string*) – how to calculate the center

Returns

- *True if all the deviances from average for all the folds*
- *are above tolerance for f1 score*
- *False if any of the deviances from the average for any of*
- *the folds are below the tolerance for f1 score*

cross_val_f1_avg(*minimum_center_tolerance*, *cv=3*, *average='binary'*, *method='mean'*)

This generates the k fold (cross validation) f1 scores, then based on computes the average of those scores. The way the average scheme works is, an average is calculated and then if the average is less than the minimum tolerance, then False is returned.

Parameters

- **minimum_center_tolerance** (*float*) – the average f1 score must be greater than this number
- **cv** (*int*) – the number of folds to consider
- **average** (*string*) – how to calculate the f1 score

- **method** (*string*) – how to calculate the center

Returns

- *True if all the folds of the f1 score are greater than*
- *the minimum_center_tolerance*
- *False if the average folds for the f1 score are less than*
- *the minimum_center_tolerance*

cross_val_f1_lower_boundary(*lower_boundary*, *cv=3*, *average='binary'*)

This is possibly the most naive stragey, it generates the k fold (cross validation) f1 scores, if any of the k folds are less than the lower boundary, then False is returned.

Parameters

- **lower_boundary** (*float*) – the lower boundary for a given f1 score
- **cv** (*int*) – the number of folds to consider
- **average** (*string*) – how to calculate the f1 score

Returns

- *True if all the folds of the f1 scores are greater than*
- *the lower_boundary*
- *False if the folds for the f1 scores are less than*
- *the lower_boundary*

cross_val_per_class_f1_anomaly_detection(*tolerance: float*, *cv=3*, *average='binary'*, *method='mean'*)

This checks the cross validated per class f1 score, based on anolamies. The way the anomaly detection scheme works is, an average is calculated and then if the deviance from the average is greater than the set tolerance, then False is returned.

Parameters

- **tolerance** (*float*) – the tolerance from the average f1 score
- **cv** (*int*) – the number of folds to consider
- **average** (*string*) – how to calculate the f1 score
- **method** (*string*) – how to calculate the center

Returns

- *True if all the deviances from average for all the folds*
- *are above tolerance for f1 score*
- *False if any of the deviances from the average for any of*
- *the folds are below the tolerance for f1 score*

cross_val_per_class_precision_anomaly_detection(*tolerance: float*, *cv=3*, *average='binary'*, *method='mean'*)

This checks the cross validated per class percision score, based on anolamies. The way the anomaly detection scheme works is, an average is calculated and then if the deviance from the average is greater than the set tolerance, then False is returned.

Parameters

- **tolerance** (*float*) – the tolerance from the average precision

- **cv** (*int*) – the number of folds to consider
- **average** (*string*) – how to calculate the precision
- **method** (*string*) – how to calculate the center

Returns

- *True if all the deviances from average for all the folds*
- *are above tolerance for precision*
- *False if any of the deviances from the average for any of*
- *the folds are below the tolerance for precision*

cross_val_per_class_recall_anomaly_detection(*tolerance: float, cv=3, average='binary', method='mean'*)

This checks the cross validated per class recall score, based on anolamies. The way the anomaly detection scheme works is, an average is calculated and then if the deviance from the average is greater than the set tolerance, then False is returned.

Parameters

- **tolerance** (*float*) – the tolerance from the average recall
- **cv** (*int*) – the number of folds to consider
- **average** (*string*) – how to calculate the recall
- **method** (*string*) – how to calculate the center

Returns

- *True if all the deviances from average for all the folds*
- *are above tolerance for recall*
- *False if any of the deviances from the average for any of*
- *the folds are below the tolerance for recall*

cross_val_per_class_roc_auc_anomaly_detection(*tolerance: float, cv=3, average='micro', method='mean'*)

This checks the cross validated per class roc auc score, based on anolamies. The way the anomaly detection scheme works is, an average is calculated and then if the deviance from the average is greater than the set tolerance, then False is returned.

Parameters

- **tolerance** (*float*) – the tolerance from the average roc auc
- **cv** (*int*) – the number of folds to consider
- **average** (*string*) – how to calculate the roc auc
- **method** (*string*) – how to calculate the center

Returns

- *True if all the deviances from average for all the folds*
- *are above tolerance for roc auc*
- *False if any of the deviances from the average for any of*
- *the folds are below the tolerance for roc auc*

cross_val_precision_anomaly_detection(*tolerance: float*, *cv=3*, *average='binary'*, *method='mean'*)

This checks the k fold (cross validation) precision score, based on anolamies. The way the anomaly detection scheme works is, an average is calculated and then if the deviance from the average is greater than the set tolerance, then False is returned.

Parameters

- **tolerance** (*float*) – the tolerance from the average precision
- **cv** (*int*) – the number of folds to consider
- **average** (*string*) – how to calculate the precision
- **method** (*string*) – how to calculate the center

Returns

- *True if all the deviances from average for all the folds*
- *are above tolerance for precision*
- *False if any of the deviances from the average for any of*
- *the folds are below the tolerance for precision*

cross_val_precision_avg(*minimum_center_tolerance*, *cv=3*, *average='binary'*, *method='mean'*)

This generates the k fold (cross validation) precision scores, then based on computes the average of those scores. The way the average scheme works is, an average is calculated and then if the average is less than the minimum tolerance, then False is returned.

Parameters

- **minimum_center_tolerance** (*float*) – the average precision must be greater than this number
- **cv** (*int*) – the number of folds to consider
- **average** (*string*) – how to calculate the precision
- **method** (*string*) – how to calculate the center

Returns

- *True if all the folds of the precision are greater than*
- *the minimum_center_tolerance*
- *False if the average folds for the precision are less than*
- *the minimum_center_tolerance*

cross_val_precision_lower_boundary(*lower_boundary*, *cv=3*, *average='binary'*)

This is possibly the most naive stragey, it generates the k fold (cross validation) precision scores, if any of the k folds are less than the lower boundary, then False is returned.

Parameters

- **lower_boundary** (*float*) – the lower boundary for a given precision score
- **cv** (*int*) – the number of folds to consider
- **average** (*string*) – how to calculate the precision

Returns

- *True if all the folds of the precision scores are*

- *greater than the lower_boundary*
- *False if the folds for the precision scores are*
- *less than the lower_boundary*

cross_val_recall_anomaly_detection(*tolerance: float*, *cv=3*, *average='binary'*, *method='mean'*)

This checks the k fold (cross validation) recall score, based on anolamies. The way the anomaly detection scheme works is, an average is calculated and then if the deviance from the average is greater than the set tolerance, then False is returned.

Parameters

- **tolerance** (*float*) – the tolerance from the average recall
- **cv** (*int*) – the number of folds to consider
- **average** (*string*) – how to calculate the recall
- **method** (*string*) – how to calculate the center

Returns

- *True if all the deviances from average for all the folds*
- *are above tolerance for recall*
- *False if any of the deviances from the average for any of*
- *the folds are below the tolerance for recall*

cross_val_recall_avg(*minimum_center_tolerance*, *cv=3*, *average='binary'*, *method='mean'*)

This generates the k fold (cross validation) recall scores, then based on computes the average of those scores. The way the average scheme works is, an average is calculated and then if the average is less than the minimum tolerance, then False is returned.

Parameters

- **minimum_center_tolerance** (*float*) – the average recall must be greater than this number
- **cv** (*int*) – the number of folds to consider
- **average** (*string*) – how to calculate the recall
- **method** (*string*) – how to calculate the center

Returns

- *True if all the folds of the recall are greater than*
- *the minimum_center_tolerance*
- *False if the average folds for the recall are less than*
- *the minimum_center_tolerance*

cross_val_recall_lower_boundary(*lower_boundary*, *cv=3*, *average='binary'*)

This is possibly the most naive stragey, it generates the k fold (cross validation) recall scores, if any of the k folds are less than the lower boundary, then False is returned.

Parameters

- **lower_boundary** (*float*) – the lower boundary for a given recall score
- **cv** (*int*) – the number of folds to consider
- **average** (*string*) – how to calculate the recall

Returns

- *True if all the folds of the recall scores are greater than*
- *the lower_boundary*
- *False if the folds for the recall scores are less than*
- *the lower_boundary*

cross_val_roc_auc_anomaly_detection(*tolerance: float*, *cv=3*, *average='micro'*, *method='mean'*)

This checks the k fold (cross validation) roc auc score, based on anolamies. The way the anomaly detection scheme works is, an average is calculated and then if the deviance from the average is greater than the set tolerance, then False is returned.

Parameters

- **tolerance** (*float*) – the tolerance from the average roc auc
- **cv** (*int*) – the number of folds to consider
- **average** (*string*) – how to calculate the roc auc
- **method** (*string*) – how to calculate the center

Returns

- *True if all the deviances from average for all the folds*
- *are above tolerance for roc auc*
- *False if any of the deviances from the average for any of*
- *the folds are below the tolerance for roc auc*

cross_val_roc_auc_avg(*minimum_center_tolerance*, *cv=3*, *average='micro'*, *method='mean'*)

This generates the k fold (cross validation) roc auc scores, then based on computes the average of those scores. The way the average scheme works is, an average is calculated and then if the average is less than the minimum tolerance, then False is returned.

Parameters

- **minimum_center_tolerance** (*float*) – the average roc auc must be greater than this number
- **cv** (*int*) – the number of folds to consider
- **average** (*string*) – how to calculate the roc auc
- **method** (*string*) – how to calculate the center

Returns

- *True if all the folds of the roc auc are greater than*
- *the minimum_center_tolerance*
- *False if the average folds for the roc auc are less than*
- *the minimum_center_tolerance*

cross_val_roc_auc_lower_boundary(*lower_boundary*, *cv=3*, *average='micro'*)

This is possibly the most naive stragey, it generates the k fold (cross validation) roc auc scores, if any of the k folds are less than the lower boundary, then False is returned.

Parameters

- **lower_boundary** (*float*) – the lower boundary for a given roc auc score

- **cv** (*int*) – the number of folds to consider
- **average** (*string*) – how to calculate the roc auc

Returns

- *True if all the folds of the roc auc scores are greater than*
- *the lower_boundary*
- *False if the folds for the roc auc scores are less than*
- *the lower_boundary*

describe_scores(*scores*, *method*)

Describes scores.

Parameters

- **scores** (*array-like*) – the scores from the model, as a list or numpy array
- **method** (*string*) – the method to use to calculate central tendency and spread

Returns

- *Returns the central tendency, and spread*
- *by method.*
- *Methods*
- *mean*
- * **central tendency** (*mean*)
- * **spread** (*standard deviation*)
- *median*
- * **central tendency** (*median*)
- * **spread** (*interquartile range*)
- *trimean*
- * **central tendency** (*trimean*)
- * **spread** (*trimean absolute deviation*)

f1_cv(*cv*, *average='binary'*)

This method performs cross-validation over f1-score.

Parameters

- **cv** (*) – The number of cross validation folds to perform
- **average** (*) – [None, 'binary'(default), 'micro', 'macro', 'samples', 'weighted'] This parameter is required for multiclass/multilabel targets. If None, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data.

  'binary': Only report results for the class specified by pos_label. This is applicable only if targets (y_{true, pred}) are binary.

  'micro': Calculate metrics globally by counting the total true positives, false negatives and false positives.

  'macro':

Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

'weighted': Calculate metrics for each label, and find their average weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance; it can result in an F-score that isnot between precision and recall.

'samples': Calculate metrics for each instance, and find their average (only meaningful for multilabel classification where this differs from accuracy_score).

Returns

Return type Returns a scores of the k-fold f1-score.

f1_lower_boundary_per_class(*lower_boundary: dict*, *average='binary'*)

This is a slightly less naive stragey, it checks the f1 score, Each class is boundary is mapped to the class via a dictionary allowing for different lower boundaries, per class. if any of the classes are less than the lower boundary, then False is returned.

Parameters

- **lower_boundary** (*dict*) – the lower boundary for each class' f1 score
- **average** (*string*) – how to calculate the f1

Returns

- *True if all the classes of the f1 scores are*
- *greater than the lower_boundary*
- *False if the classes for the f1 scores are*
- *less than the lower_boundary*

get_test_score(*cross_val_dict*)

is_binary()

If number of classes == 2 returns True False otherwise

precision_cv(*cv*, *average='binary'*)

This method performs cross-validation over precision.

Parameters

- **cv** (*) – The number of cross validation folds to perform
- **average** (*) – [None, 'binary'(default), 'micro', 'macro', 'samples', 'weighted'] This parameter is required for multiclass/multilabel targets. If None, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data.

  'binary': Only report results for the class specified by pos_label. This is applicable only if targets (y_{true, pred}) are binary.

  'micro': Calculate metrics globally by counting the total true positives, false negatives and false positives.

  'macro':

  Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

'weighted': Calculate metrics for each label, and find their average weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance; it can result in an F-score that isnot between precision and recall.

'samples': Calculate metrics for each instance, and find their average (only meaningful for multilabel classification where this differs from accuracy_score).

Returns

Return type Returns a scores of the k-fold precision.

precision_lower_boundary_per_class(*lower_boundary: dict*, *average='binary'*)

This is a slightly less naive stragey, it checks the precision score, Each class is boundary is mapped to the class via a dictionary allowing for different lower boundaries, per class. if any of the classes are less than the lower boundary, then False is returned.

Parameters

- **lower_boundary** (*dict*) – the lower boundary for each class' precision score
- **average** (*string*) – how to calculate the precision

Returns

- *True if all the classes of the precision scores are*
- *greater than the lower_boundary*
- *False if the classes for the precision scores are*
- *less than the lower_boundary*

recall_cv(*cv*, *average='binary'*)

This method performs cross-validation over recall.

Parameters

- **cv** (*) – The number of cross validation folds to perform
- **average** (*) – [None, 'binary'(default), 'micro', 'macro', 'samples', 'weighted'] This parameter is required for multiclass/multilabel targets. If None, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data.

  'binary': Only report results for the class specified by pos_label. This is applicable only if targets (y_{true, pred}) are binary.

  'micro': Calculate metrics globally by counting the total true positives, false negatives and false positives.

  'macro':

  Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

  'weighted': Calculate metrics for each label, and find their average weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance; it can result in an F-score that isnot between precision and recall.

  'samples': Calculate metrics for each instance, and find their average (only meaningful for multilabel classification where this differs from accuracy_score).

Returns

Return type Returns a scores of the k-fold recall.

recall_lower_boundary_per_class(*lower_boundary: dict*, *average='binary'*)
This is a slightly less naive stragey, it checks the recall score, Each class is boundary is mapped to the class via a dictionary allowing for different lower boundaries, per class. if any of the classes are less than the lower boundary, then False is returned.

Parameters

- **lower_boundary** (*dict*) – the lower boundary for each class' recall score
- **average** (*string*) – how to calculate the recall

Returns

- *True if all the classes of the recall scores are*
- *greater than the lower_boundary*
- *False if the classes for the recall scores are*
- *less than the lower_boundary*

reset_average(*average*)
Resets the average to the correct thing. If the number of classes are not binary, Then average is changed to micro. Otherwise, return the current average.

roc_auc_cv(*cv*, *average='micro'*)
This method performs cross-validation over roc_auc.

Parameters

- **cv** (*) – The number of cross validation folds to perform
- **average** (*) – [None, 'binary'(default), 'micro', 'macro', 'samples', 'weighted'] This parameter is required for multiclass/multilabel targets. If None, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data.

  'binary': Only report results for the class specified by pos_label. This is applicable only if targets (y_{true, pred}) are binary.

  'micro': Calculate metrics globally by counting the total true positives, false negatives and false positives.

  'macro':

  Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

  'weighted': Calculate metrics for each label, and find their average weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance; it can result in an F-score that isnot between precision and recall.

  'samples': Calculate metrics for each instance, and find their average (only meaningful for multilabel classification where this differs from accuracy_score).

Returns

Return type Returns a scores of the k-fold roc_auc.

roc_auc_exception()
Ensures roc_auc score is used correctly. ROC AUC is only defined for binary classification.

roc_auc_lower_boundary_per_class(*lower_boundary: dict*, *average='micro'*)
This is a slightly less naive stragey, it checks the roc auc score, Each class is boundary is mapped to the class via a dictionary allowing for different lower boundaries, per class. if any of the classes are less than the lower boundary, then False is returned.

Parameters

- **lower_boundary** (*dict*) – the lower boundary for each class' roc auc score
- **average** (*string*) – how to calculate the roc auc

Returns

- *True if all the classes of the roc auc scores are*
- *greater than the lower_boundary*
- *False if the classes for the roc auc scores are*
- *less than the lower_boundary*

run_energy_stress_test(*sample_sizes: list*, *max_energy_usages: list*, *print_to_screen=False*, *print_to_pdf=False*)
This is a performance test to ensure that the model is energy efficient.

Note: the model must take longer than 5 seconds to run otherwise energyusage cannot accurate estimate the energy cost. At this point, the cost is neglible. Therefore, when testing, please try to use reasonable size estimates based on expected throughput.

Parameters

- **sample_sizes** (*list*) – the size of each sample to test for doing a prediction, each sample size is an integer
- **max_energy_usages** (*list*) – the maximum time in seconds that each sample should take to predict, at a maximum.

Returns

- *True if all samples predict within the maximum allowed*
- *energy usage.*
- *False otherwise.*

run_time_stress_test(*sample_sizes: list*, *max_run_times: list*)
This is a performance test to ensure that the model runs fast enough.

sample_sizes [list] the size of each sample to test for doing a prediction, each sample size is an integer

max_run_times [list] the maximum time in seconds that each sample should take to predict, at a maximum.

Returns

- *True if all samples predict within the maximum allowed*
- *time.*
- *False otherwise.*

spread_cross_val_classifier_testing(*precision_tolerance: float*, *recall_tolerance: float*, *f1_tolerance: float*, *method='mean'*, *cv=10*, *average='binary'*)

This is a somewhat intelligent stragey, it generates the k fold (cross validation) the following scores: * precision scores, * recall scores * f1 scores if any of the k folds score less than the center - (spread * tolerance), then False is returned.

Parameters

- **tolerance** (*float*) – the tolerance modifier for how far below the center the score can be before a false is returned
- **method** (*string*) – see describe for more details. * mean : the center is the mean, the spread is standard

  deviation.

  - **median** [the center is the median, the spread is] the interquartile range.
  - **trimean** [the center is the trimean, the spread is] trimean absolute deviation.
- **average** (*string*) – how to calculate the precision

Returns

- *True if all the folds of the precision, recall, f1 scores*
- *are greater than the center - (spread * tolerance)*
- *False if the folds for the precision, recall, f1 scores*
- *are less than the center - (spread * tolerance)*

spread_cross_val_f1_anomaly_detection(*tolerance*, *method='mean'*, *cv=10*, *average='binary'*)

This is a somewhat intelligent stragey, it generates the k fold (cross validation) f1 scores, if any of the k folds score less than the center - (spread * tolerance), then False is returned.

Parameters

- **tolerance** (*float*) – the tolerance modifier for how far below the center the score can be before a false is returned
- **method** (*string*) – see describe for more details. * mean : the center is the mean, the spread is standard

  deviation.

  - **median** [the center is the median, the spread is] the interquartile range.
  - **trimean** [the center is the trimean, the spread is] trimean absolute deviation.
- **average** (*string*) – how to calculate the f1 score

Returns

- *True if all the folds of the f1 scores are greater than*
- *the center - (spread * tolerance)*
- *False if the folds for the f1 scores are less than*
- *the center - (spread * tolerance)*

spread_cross_val_precision_anomaly_detection(*tolerance*, *method='mean'*, *cv=10*, *average='binary'*)

This is a somewhat intelligent stragey, it generates the k fold (cross validation) precision scores, if any of the k folds score less than the center - (spread * tolerance), then False is returned.

Parameters

- **tolerance** (*float*) – the tolerance modifier for how far below the center the score can be before a false is returned
- **method** (*string*) – see describe for more details. * mean : the center is the mean, the spread is standard

  deviation.

  - **median** [the center is the median, the spread is] the interquartile range.
  - **trimean** [the center is the trimean, the spread is] trimean absolute deviation.
- **average** (*string*) – how to calculate the precision

Returns

- *True if all the folds of the precision scores are greater than*
- *the center - (spread * tolerance)*
- *False if the folds for the precision scores are less than*
- *the center - (spread * tolerance)*

spread_cross_val_recall_anomaly_detection(*tolerance*, *method='mean'*, *cv=3*, *average='binary'*)

This is a somewhat intelligent stragey, it generates the k fold (cross validation) recall scores, if any of the k folds score less than the center - (spread * tolerance), then False is returned.

Parameters

- **tolerance** (*float*) – the tolerance modifier for how far below the center the score can be before a false is returned
- **method** (*string*) – see describe for more details. * mean : the center is the mean, the spread is standard

  deviation.

  - **median** [the center is the median, the spread is] the interquartile range.
  - **trimean** [the center is the trimean, the spread is] trimean absolute deviation.
- **average** (*string*) – how to calculate the recall

Returns

- *True if all the folds of the recall scores are greater than*
- *the center - (spread * tolerance)*
- *False if the folds for the recall scores are less than*
- *the center - (spread * tolerance)*

spread_cross_val_roc_auc_anomaly_detection(*tolerance*, *method='mean'*, *cv=10*, *average='micro'*)

This is a somewhat intelligent stragey, it generates the k fold (cross validation) roc auc scores, if any of the k folds score less than the center - (spread * tolerance), then False is returned.

Parameters

- **tolerance** (*float*) – the tolerance modifier for how far below the center the score can be before a false is returned
- **method** (*string*) – see describe for more details. * mean : the center is the mean, the spread is standard

  deviation.

  - **median** [the center is the median, the spread is] the interquartile range.
  - **trimean** [the center is the trimean, the spread is] trimean absolute deviation.
- **average** (*string*) – how to calculate the precision

Returns

- *True if all the folds of the roc auc scores are greater than*
- *the center - (spread * tolerance)*
- *False if the folds for the roc auc scores are less than*
- *the center - (spread * tolerance)*

trimean(*data*)

I'm exposing this as a public method because the trimean is not implemented in enough packages.

Formula: (25th percentile + 2*50th percentile + 75th percentile)/4

Parameters **data** (*array-like*) – an iterable, either a list or a numpy array

Returns **the trimean**

Return type float

trimean_absolute_deviation(*data*)

The trimean absolute deviation is the the average distance from the trimean.

Parameters **data** (*array-like*) – an iterable, either a list or a numpy array

Returns **the average distance to the trimean**

Return type float

class drifter_ml.classification_tests.**ClassifierComparison**(*clf_one*, *clf_two*, *test_data*, *target_name*, *column_names*)

Bases: *drifter_ml.classification_tests.classification_tests.FixedClassificationMetrics*

cross_val_f1(*clf*, *cv=3*, *average='binary'*)

cross_val_f1_per_class(*clf*, *cv=3*, *average='binary'*)

cross_val_per_class_two_model_classifier_testing(*cv=3*, *average='binary'*)

cross_val_precision(*clf*, *cv=3*, *average='binary'*)

cross_val_precision_per_class(*clf*, *cv=3*, *average='binary'*)

cross_val_recall(*clf*, *cv=3*, *average='binary'*)

cross_val_recall_per_class(*clf*, *cv=3*, *average='binary'*)

cross_val_roc_auc(*clf*, *cv=3*, *average='micro'*)

cross_val_roc_auc_per_class(*clf*, *cv=3*, *average='micro'*)

cross_val_two_model_classifier_testing(*cv=3*, *average='binary'*)

f1_per_class(*clf*, *average='binary'*)

is_binary()

precision_per_class(*clf*, *average='binary'*)

recall_per_class(*clf*, *average='binary'*)

reset_average(*average*)

roc_auc_exception()

roc_auc_per_class(*clf*, *average='micro'*)

two_model_classifier_testing(*average='binary'*)

two_model_prediction_run_time_stress_test(*sample_sizes*)

CHAPTER 7

drifter_ml.regression_tests package

7.1 Submodules

7.2 drifter_ml.regression_tests.regression_tests module

class drifter_ml.regression_tests.regression_tests.**RegressionComparison**(*reg_one*, *reg_two*, *test_data*, *target_name*, *column_names*)

Bases: object

cross_val_mae_result(*reg*, *cv=3*)

cross_val_mse_result(*reg*, *cv=3*)

cv_two_model_regression_testing(*cv=3*)

mae_result(*reg*)

mse_result(*reg*)

two_model_prediction_run_time_stress_test(*sample_sizes*)

two_model_regression_testing()

class drifter_ml.regression_tests.regression_tests.**RegressionTests**(*reg*, *test_data*, *target_name*, *column_names*)

Bases: object

cross_val_mae_anomaly_detection(*tolerance*, *cv=3*, *method='mean'*)

cross_val_mae_avg(*minimum_center_tolerance*, *cv=3*, *method='mean'*)

cross_val_mae_upper_boundary(*upper_boundary*, *cv=3*)

cross_val_mse_anomaly_detection(*tolerance*, *cv=3*, *method='mean'*)

cross_val_mse_avg(*minimum_center_tolerance*, *cv=3*, *method='mean'*)

cross_val_mse_upper_boundary(*upper_boundary*, *cv=3*)

cross_val_tae_anomaly_detection(*tolerance*, *cv=3*, *method='mean'*)

cross_val_tae_avg(*minimum_center_tolerance*, *cv=3*, *method='mean'*)

cross_val_tae_upper_boundary(*upper_boundary*, *cv=3*)

cross_val_tse_anomaly_detection(*tolerance*, *cv=3*, *method='mean'*)

cross_val_tse_avg(*minimum_center_tolerance*, *cv=3*, *method='mean'*)

cross_val_tse_upper_boundary(*upper_boundary*, *cv=3*)

describe_scores(*scores*, *method*)

Describes scores.

Parameters

- **scores** (*array-like*) – the scores from the model, as a list or numpy array
- **method** (*string*) – the method to use to calculate central tendency and spread

Returns

- *Returns the central tendency, and spread*
- *by method.*
- *Methods*
- *mean*
- *** central tendency** (*mean*)
- *** spread** (*standard deviation*)
- *median*
- *** central tendency** (*median*)
- *** spread** (*interquartile range*)
- *trimean*
- *** central tendency** (*trimean*)
- *** spread** (*trimean absolute deviation*)

get_test_score(*cross_val_dict*)

mae_cv(*cv*)

This method performs cross-validation over median absolute error.

Parameters **cv** (*) – The number of cross validation folds to perform

Returns

Return type Returns a scores of the k-fold median absolute error.

mae_upper_boundary(*upper_boundary*)

mse_cv(*cv*)
This method performs cross-validation over mean squared error.

Parameters **cv** (*) – The number of cross validation folds to perform

Returns

Return type Returns a scores of the k-fold mean squared error.

mse_upper_boundary(*upper_boundary*)

run_time_stress_test(*sample_sizes*, *max_run_times*)

tae_cv(*cv*)
This method performs cross-validation over trimean absolute error.

Parameters **cv** (*) – The number of cross validation folds to perform

Returns

Return type Returns a scores of the k-fold trimean absolute error.

tae_upper_boundary(*upper_boundary*)

trimean(*data*)
I'm exposing this as a public method because the trimean is not implemented in enough packages.

Formula: (25th percentile + 2*50th percentile + 75th percentile)/4

Parameters **data** (*array-like*) – an iterable, either a list or a numpy array

Returns **the trimean**

Return type float

trimean_absolute_deviation(*data*)
The trimean absolute deviation is the the average distance from the trimean.

Parameters **data** (*array-like*) – an iterable, either a list or a numpy array

Returns **the average distance to the trimean**

Return type float

trimean_absolute_error(*y_true*, *y_pred*, *sample_weight=None*, *multioutput='uniform_average'*)

trimean_squared_error(*y_true*, *y_pred*, *sample_weight=None*, *multioutput='uniform_average'*)

tse_cv(*cv*)
This method performs cross-validation over trimean squared error.

Parameters **cv** (*) – The number of cross validation folds to perform

Returns

Return type Returns a scores of the k-fold trimean squared error.

tse_upper_boundary(*upper_boundary*)

upper_bound_regression_testing(*mse_upper_boundary*, *mae_upper_boundary*, *tse_upper_boundary*, *tae_upper_boundary*)

7.3 Module contents

class drifter_ml.regression_tests.**RegressionTests**(*reg*, *test_data*, *target_name*, *column_names*)

Bases: object

cross_val_mae_anomaly_detection(*tolerance*, *cv=3*, *method='mean'*)

cross_val_mae_avg(*minimum_center_tolerance*, *cv=3*, *method='mean'*)

cross_val_mae_upper_boundary(*upper_boundary*, *cv=3*)

cross_val_mse_anomaly_detection(*tolerance*, *cv=3*, *method='mean'*)

cross_val_mse_avg(*minimum_center_tolerance*, *cv=3*, *method='mean'*)

cross_val_mse_upper_boundary(*upper_boundary*, *cv=3*)

cross_val_tae_anomaly_detection(*tolerance*, *cv=3*, *method='mean'*)

cross_val_tae_avg(*minimum_center_tolerance*, *cv=3*, *method='mean'*)

cross_val_tae_upper_boundary(*upper_boundary*, *cv=3*)

cross_val_tse_anomaly_detection(*tolerance*, *cv=3*, *method='mean'*)

cross_val_tse_avg(*minimum_center_tolerance*, *cv=3*, *method='mean'*)

cross_val_tse_upper_boundary(*upper_boundary*, *cv=3*)

describe_scores(*scores*, *method*)

Describes scores.

Parameters

- **scores** (*array-like*) – the scores from the model, as a list or numpy array
- **method** (*string*) – the method to use to calculate central tendency and spread

Returns

- *Returns the central tendency, and spread*
- *by method.*
- *Methods*
- *mean*
- * **central tendency** (*mean*)
- * **spread** (*standard deviation*)
- *median*
- * **central tendency** (*median*)
- * **spread** (*interquartile range*)
- *trimean*
- * **central tendency** (*trimean*)
- * **spread** (*trimean absolute deviation*)

get_test_score(*cross_val_dict*)

mae_cv(*cv*)

This method performs cross-validation over median absolute error.

Parameters **cv** (*) – The number of cross validation folds to perform

Returns

Return type Returns a scores of the k-fold median absolute error.

mae_upper_boundary(*upper_boundary*)

mse_cv(*cv*)
This method performs cross-validation over mean squared error.

Parameters **cv** (*) – The number of cross validation folds to perform

Returns

Return type Returns a scores of the k-fold mean squared error.

mse_upper_boundary(*upper_boundary*)

run_time_stress_test(*sample_sizes*, *max_run_times*)

tae_cv(*cv*)
This method performs cross-validation over trimean absolute error.

Parameters **cv** (*) – The number of cross validation folds to perform

Returns

Return type Returns a scores of the k-fold trimean absolute error.

tae_upper_boundary(*upper_boundary*)

trimean(*data*)
I'm exposing this as a public method because the trimean is not implemented in enough packages.

Formula: (25th percentile + 2*50th percentile + 75th percentile)/4

Parameters **data** (*array-like*) – an iterable, either a list or a numpy array

Returns **the trimean**

Return type float

trimean_absolute_deviation(*data*)
The trimean absolute deviation is the the average distance from the trimean.

Parameters **data** (*array-like*) – an iterable, either a list or a numpy array

Returns **the average distance to the trimean**

Return type float

trimean_absolute_error(*y_true*, *y_pred*, *sample_weight=None*, *multioutput='uniform_average'*)

trimean_squared_error(*y_true*, *y_pred*, *sample_weight=None*, *multioutput='uniform_average'*)

tse_cv(*cv*)
This method performs cross-validation over trimean squared error.

Parameters **cv** (*) – The number of cross validation folds to perform

Returns

Return type Returns a scores of the k-fold trimean squared error.

tse_upper_boundary(*upper_boundary*)

upper_bound_regression_testing(*mse_upper_boundary*, *mae_upper_boundary*, *tse_upper_boundary*, *tae_upper_boundary*)

class drifter_ml.regression_tests.**RegressionComparison**(*reg_one*, *reg_two*, *test_data*, *target_name*, *column_names*)

Bases: object

cross_val_mae_result(*reg*, *cv=3*)

cross_val_mse_result(*reg*, *cv=3*)

cv_two_model_regression_testing(*cv=3*)

mae_result(*reg*)

mse_result(*reg*)

two_model_prediction_run_time_stress_test(*sample_sizes*)

two_model_regression_testing()

CHAPTER 8

drifter_ml.columnar_tests package

8.1 Submodules

8.2 drifter_ml.columnar_tests.columnar_tests module

class drifter_ml.columnar_tests.columnar_tests.**ColumnarData**(*historical_data*, *new_data*)

Bases: object

is_normal(*column*)

kruskal_similar_distribution(*column*, *pvalue_threshold=0.05*, *num_rounds=3*)

ks_2samp_similar_distribution(*column*, *pvalue_threshold=0.05*, *num_rounds=3*)

mann_whitney_u_similar_distribution(*column*, *pvalue_threshold=0.05*, *num_rounds=3*)

mean_similarity(*column*, *tolerance=2*)

median_similarity(*column*, *tolerance=2*)

pearson_similar_correlation(*column*, *correlation_lower_bound*, *pvalue_threshold=0.05*, *num_rounds=3*)

spearman_similar_correlation(*column*, *correlation_lower_bound*, *pvalue_threshold=0.05*, *num_rounds=3*)

trimean(*data*)

trimean_absolute_deviation(*data*)

trimean_similarity(*column*, *tolerance=2*)

wilcoxon_similar_distribution(*column*, *pvalue_threshold=0.05*, *num_rounds=3*)

class drifter_ml.columnar_tests.columnar_tests.**DataSanitization**(*data*)

Bases: object

has_completeness(*column*, *threshold*)

has_uniqueness(*column*, *threshold*)

is_complete(*column*)

is_in_range(*column*, *lower_bound*, *upper_bound*, *threshold*)

is_less_than(*column_one*, *column_two*)

is_non_negative(*column*)

is_unique(*column*)

8.3 Module contents

class drifter_ml.columnar_tests.**DataSanitization**(*data*)

Bases: object

has_completeness(*column*, *threshold*)

has_uniqueness(*column*, *threshold*)

is_complete(*column*)

is_in_range(*column*, *lower_bound*, *upper_bound*, *threshold*)

is_less_than(*column_one*, *column_two*)

is_non_negative(*column*)

is_unique(*column*)

class drifter_ml.columnar_tests.**ColumnarData**(*historical_data*, *new_data*)

Bases: object

is_normal(*column*)

kruskal_similar_distribution(*column*, *pvalue_threshold=0.05*, *num_rounds=3*)

ks_2samp_similar_distribution(*column*, *pvalue_threshold=0.05*, *num_rounds=3*)

mann_whitney_u_similar_distribution(*column*, *pvalue_threshold=0.05*, *num_rounds=3*)

mean_similarity(*column*, *tolerance=2*)

median_similarity(*column*, *tolerance=2*)

pearson_similar_correlation(*column*, *correlation_lower_bound*, *pvalue_threshold=0.05*, *num_rounds=3*)

spearman_similar_correlation(*column*, *correlation_lower_bound*, *pvalue_threshold=0.05*, *num_rounds=3*)

trimean(*data*)

trimean_absolute_deviation(*data*)

trimean_similarity(*column*, *tolerance=2*)

wilcoxon_similar_distribution(*column*, *pvalue_threshold=0.05*, *num_rounds=3*)

CHAPTER 9

drifter_ml.structural_tests package

9.1 Submodules

9.2 drifter_ml.structural_tests.structural_tests module

class drifter_ml.structural_tests.structural_tests.**DBscanClustering**(*new_data*, *historical_data*, *column_names*, *target_name*)

Bases: object

adjusted_rand_dbscan_scorer(*min_similarity*)

completeness_dbscan_scorer(*min_similarity*)

dbscan_clusters(*data*)

dbscan_scorer(*metric*, *min_similarity*)

fowlkes_mallows_dbscan_scorer(*min_similarity*)

homogeneity_dbscan_scorer(*min_similarity*)

mutual_info_dbscan_scorer(*min_similarity*)

unsupervised_dbscan_score_clustering(*min_similarity*)

v_measure_dbscan_scorer(*min_similarity*)

class drifter_ml.structural_tests.structural_tests.**KmeansClustering**(*new_data*, *historical_data*, *column_names*, *target_name*)

Bases: object

adjusted_rand_kmeans_scorer(*min_similarity*)

completeness_kmeans_scorer(*min_similarity*)

fowlkes_mallows_kmeans_scorer(*min_similarity*)

homogeneity_kmeans_scorer(*min_similarity*)

kmeans_clusters(*n_clusters*, *data*)

kmeans_scorer(*metric*, *min_similarity*)

mutual_info_kmeans_scorer(*min_similarity*)

unsupervised_kmeans_score_clustering(*min_similarity*)

v_measure_kmeans_scorer(*min_similarity*)

class drifter_ml.structural_tests.structural_tests.**KnnClustering**(*new_data*, *historical_data*, *column_names*, *target_name*)

Bases: object

cls_supervised_clustering(*data*)

cls_supervised_similar_clustering(*absolute_distance*)

reg_supervised_clustering(*data*)

reg_supervised_similar_clustering(*absolute_distance*)

class drifter_ml.structural_tests.structural_tests.**StructuralData**(*new_data*, *historical_data*, *column_names*, *target_name*)

Bases: *drifter_ml.structural_tests.structural_tests.KnnClustering*, *drifter_ml.structural_tests.structural_tests.DBscanClustering*, *drifter_ml.structural_tests.structural_tests.KmeansClustering*

9.3 Module contents

class drifter_ml.structural_tests.**StructuralData**(*new_data*, *historical_data*, *column_names*, *target_name*)

Bases: *drifter_ml.structural_tests.structural_tests.KnnClustering*,

drifter_ml.structural_tests.structural_tests.DBscanClustering, *drifter_ml.structural_tests.structural_tests.KmeansClustering*

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

D

F

G

H

I

K

M

P

R

U

V

W