CHAPTER 9
Hyper-Parameter Tuning with Cross-Validation

9.1 MOTIVATION

Hyper-parameter tuning is an essential step in fitting an ML algorithm. When this is not done properly, the algorithm is likely to overfit, and live performance will disappoint. The ML literature places special attention on cross-validating any tuned hyper-parameter. As we have seen in Chapter 7, cross-validation (CV) in finance is an especially difficult problem, where solutions from other fields are likely to fail. In this chapter we will discuss how to tune hyper-parameters using the purged k-fold CV method. The references section lists studies that propose alternative methods that may be useful in specific problems.

9.2 GRID SEARCH CROSS-VALIDATION

Grid search cross-validation conducts an exhaustive search for the combination of parameters that maximizes the CV performance, according to some user-defined score function. When we do not know much about the underlying structure of the data, this is a reasonable first approach. Scikit-learn has implemented this logic in the function `GridSearchCV`, which accepts a CV generator as an argument. For the reasons explained in Chapter 7, we need to pass our `PurgedKFold` class (Snippet 7.3) in order to prevent that `GridSearchCV` overfits the ML estimator to leaked information.

SNIPPET 9.1 GRID SEARCH WITH PURGED K-FOLD CROSS-VALIDATION

```
def clfHyperFit(feat,lbl,t1,pipe_clf,param_grid,cv=3,bagging=[0,None,1.],
                n_jobs=-1,pctEmbargo=0,**fit_params):
    if set(lbl.values)=={0,1}:scoring='f1' # f1 for meta-labeling
    else:scoring='neg_log_loss' # symmetric towards all cases
    #1) hyperparameter search, on train data
    inner_cv=PurgedKFold(n_splits=cv,t1=t1,pctEmbargo=pctEmbargo) # purged
    gs=GridSearchCV(estimator=pipe_clf,param_grid=param_grid,
        scoring=scoring,cv=inner_cv,n_jobs=n_jobs,iid=False)
    gs=gs.fit(feat,lbl,**fit_params).best_estimator_ # pipeline
    #2) fit validated model on the entirety of the data
    if bagging[1]>0:
        gs=BaggingClassifier(base_estimator=MyPipeline(gs.steps),
            n_estimators=int(bagging[0]),max_samples=float(bagging[1]),
            max_features=float(bagging[2]),n_jobs=n_jobs)
        gs=gs.fit(feat,lbl,sample_weight=fit_params \
            [gs.base_estimator.steps[-1][0]+'__sample_weight'])
        gs=Pipeline([('bag',gs)])
    return gs
```

Snippet 9.1 lists function `clfHyperFit`, which implements a purged `GridSearchCV`. The argument `fit_params` can be used to pass `sample_weight`, and `param_grid` contains the values that will be combined into a grid. In addition, this function allows for the bagging of the tuned estimator. Bagging an estimator is generally a good idea for the reasons explained in Chapter 6, and the above function incorporates logic to that purpose.

I advise you to use `scoring=‘f1’` in the context of meta-labeling applications, for the following reason. Suppose a sample with a very large number of negative (i.e., label ‘0’) cases. A classifier that predicts all cases to be negative will achieve high `'accuracy'` or `'neg_log_loss'`, even though it has not learned from the features how to discriminate between cases. In fact, such a model achieves zero recall and undefined precision (see Chapter 3, Section 3.7). The `'f1'` score corrects for that performance inflation by scoring the classifier in terms of precision and recall (see Chapter 14, Section 14.8).

For other (non-meta-labeling) applications, it is fine to use `'accuracy'` or `'neg_log_loss'`, because we are equally interested in predicting all cases. Note that a relabeling of cases has no impact on `'accuracy'` or `'neg_log_loss'`, however it will have an impact on `'f1'`.

This example introduces nicely one limitation of sklearn’s `Pipelines` : Their fit method does not expect a `sample_weight` argument. Instead, it expects a `fit_params` keyworded argument. That is a bug that has been reported in

GitHub; however, it may take some time to fix it, as it involves rewriting and testing much functionality. Until then, feel free to use the workaround in Snippet 9.2. It creates a new class, called `MyPipeline`, which inherits all methods from sklearn's `Pipeline`. It overwrites the inherited `fit` method with a new one that handles the argument `sample_weight`, after which it redirects to the parent class.

SNIPPET 9.2 AN ENHANCED PIPELINE CLASS

```
class MyPipeline(Pipeline):
    def fit(self,X,y,sample_weight=None,**fit_params):
        if sample_weight is not None:
            fit_params[self.steps[-1][0]+'__sample_weight']=sample_weight
        return super(MyPipeline,self).fit(X,y,**fit_params)
```

If you are not familiar with this technique for expanding classes, you may want to read this introductory Stackoverflow post: http://stackoverflow.com/questions/576169/understanding-python-super-with-init-methods.

9.3 RANDOMIZED SEARCH CROSS-VALIDATION

For ML algorithms with a large number of parameters, a grid search cross-validation (CV) becomes computationally intractable. In this case, an alternative with good statistical properties is to sample each parameter from a distribution (Begstra et al. [2011, 2012]). This has two benefits: First, we can control for the number of combinations we will search for, regardless of the dimensionality of the problem (the equivalent to a computational budget). Second, having parameters that are relatively irrelevant performance-wise will not substantially increase our search time, as would be the case with grid search CV.

Rather than writing a new function to work with `RandomizedSearchCV`, let us expand Snippet 9.1 to incorporate an option to this purpose. A possible implementation is Snippet 9.3.

SNIPPET 9.3 RANDOMIZED SEARCH WITH PURGED K-FOLD CV

```
def clfHyperFit(feat,lbl,t1,pipe_clf,param_grid,cv=3,bagging=[0,None,1.],
                rndSearchIter=0,n_jobs=-1,pctEmbargo=0,**fit_params):
    if set(lbl.values)=={0,1}:scoring='f1' # f1 for meta-labeling
    else:scoring='neg_log_loss' # symmetric towards all cases
    #1) hyperparameter search, on train data
    inner_cv=PurgedKFold(n_splits=cv,t1=t1,pctEmbargo=pctEmbargo) # purged
    if rndSearchIter==0:
        gs=GridSearchCV(estimator=pipe_clf,param_grid=param_grid,
            scoring=scoring,cv=inner_cv,n_jobs=n_jobs,iid=False)
    else:
        gs=RandomizedSearchCV(estimator=pipe_clf,param_distributions= \
            param_grid,scoring=scoring,cv=inner_cv,n_jobs=n_jobs,
            iid=False,n_iter=rndSearchIter)
    gs=gs.fit(feat,lbl,**fit_params).best_estimator_ # pipeline
    #2) fit validated model on the entirety of the data
    if bagging[1]>0:
        gs=BaggingClassifier(base_estimator=MyPipeline(gs.steps),
            n_estimators=int(bagging[0]),max_samples=float(bagging[1]),
            max_features=float(bagging[2]),n_jobs=n_jobs)
        gs=gs.fit(feat,lbl,sample_weight=fit_params \
            [gs.base_estimator.steps[-1][0]+'__sample_weight'])
        gs=Pipeline([('bag',gs)])
    return gs
```

9.3.1 Log-Uniform Distribution

It is common for some ML algorithms to accept non-negative hyper-parameters only. That is the case of some very popular parameters, such as `C` in the SVC classifier and `gamma` in the RBF kernel.[1] We could draw random numbers from a uniform distribution bounded between 0 and some large value, say 100. That would mean that 99% of the values would be expected to be greater than 1. That is not necessarily the most effective way of exploring the feasibility region of parameters whose functions do not respond linearly. For example, an SVC can be as responsive to an increase in `C` from 0.01 to 1 as to an increase in `C` from 1 to 100.[2] So sampling `C` from a $U[0, 100]$ (uniform) distribution will be inefficient. In those instances, it seems more effective to draw values from a distribution where the logarithm of those draws will be distributed uniformly. I call that a “log-uniform distribution,” and since I could not find it in the literature, I must define it properly.

A random variable x follows a log-uniform distribution between $a > 0$ and $b > a$ if and only if $\log[x] \sim U[\log[a], \log[b]]$. This distribution has a CDF:

$$F[x] = \begin{cases} \dfrac{\log[x] - \log[a]}{\log[b] - \log[a]} & \text{for } a \le x \le b \\ 0 & \text{for } x < a \\ 1 & \text{for } x > b \end{cases}$$

From this, we derive a PDF:

$$f[x] = \begin{cases} \dfrac{1}{x \log\left[b/a\right]} & \text{for } a \le x \le b \\ 0 & \text{for } x < a \\ 0 & \text{for } x > b \end{cases}$$

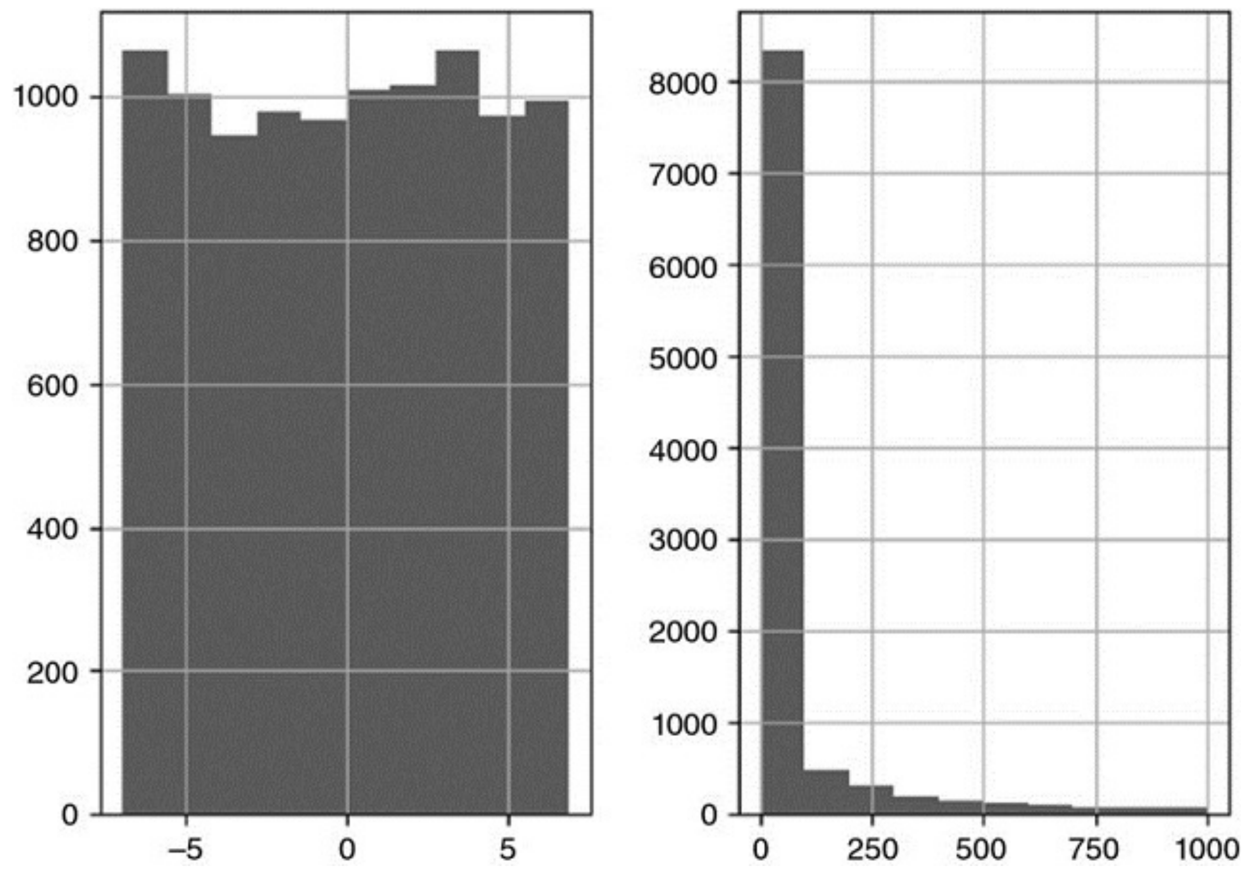

FIGURE 9.1 Result from testing the `logUniform_gen` class

Note that the CDF is invariant to the base of the logarithm, since $\frac{\log\left[\frac{x}{a}\right]}{\log\left[\frac{b}{a}\right]} = \frac{\log_c\left[\frac{x}{a}\right]}{\log_c\left[\frac{b}{a}\right]}$ for any base c, thus the random variable is not a function of c. Snippet 9.4 implements (and tests) in `scipy.stats` a random variable where $[a, b] = [1E-3, 1E3]$, hence $\log[x] \sim U[\log[1E-3], \log[1E3]]$. Figure 9.1 illustrates the uniformity of the samples in log-scale.

SNIPPET 9.4 THE `LOGUNIFORM_GEN` CLASS

```
import numpy as np,pandas as pd,matplotlib.pyplot as mpl
from scipy.stats import rv_continuous,kstest
#———————————————————————————————————————
class logUniform_gen(rv_continuous):
    # random numbers log-uniformly distributed between 1 and e
    def _cdf(self,x):
        return np.log(x/self.a)/np.log(self.b/self.a)
def logUniform(a=1,b=np.exp(1)):return logUniform_gen(a=a,b=b,name='logUniform')
#———————————————————————————————————————
a,b,size=1E-3,1E3,10000
vals=logUniform(a=a,b=b).rvs(size=size)
print kstest(rvs=np.log(vals),cdf='uniform',args=(np.log(a),np.log(b/a)),N=size)
print pd.Series(vals).describe()
mpl.subplot(121)
pd.Series(np.log(vals)).hist()
mpl.subplot(122)
pd.Series(vals).hist()
mpl.show()
```

9.4 SCORING AND HYPER-PARAMETER TUNING

Snippets 9.1 and 9.3 set `scoring='f1'` for meta-labeling applications. For other applications, they set `scoring='neg_log_loss'` rather than the standard `scoring='accuracy'`. Although accuracy has a more intuitive interpretation, I suggest that you use `neg_log_loss` when you are tuning hyper-parameters for an investment strategy. Let me explain my reasoning.

Suppose that your ML investment strategy predicts that you should buy a security, with high probability. You will enter a large long position, as a function of the strategy's confidence. If the prediction was erroneous, and the market sells off instead, you will lose a lot of money. And yet, accuracy accounts equally for an erroneous buy prediction with high probability and for an erroneous buy prediction with low probability. Moreover, accuracy can offset a miss with high probability with a hit with low probability.

Investment strategies profit from predicting the right label with high confidence. Gains from good predictions with low confidence will not suffice to offset the losses from bad predictions with high confidence. For this reason, accuracy does not provide a realistic scoring of the classifier's performance. Conversely, log loss[3] (aka cross-entropy loss) computes the log-likelihood of the classifier given the true label, which takes predictions' probabilities into account. Log loss can be estimated as follows:

$$L[Y,P] = -\log[\mathrm{Prob}[Y|P]] = -N^{-1}\sum_{n=0}^{N-1}\sum_{k=0}^{K-1} y_{n,k}\log\left[p_{n,k}\right]$$

where

- $p_{n,k}$ is the probability associated with prediction *n* of label *k*.
- *Y* is a 1-of-*K* binary indicator matrix, such that $y_{n,k} = 1$ when observation *n* was assigned label *k* out of *K* possible labels, and 0 otherwise.

Suppose that a classifier predicts two 1s, where the true labels are 1 and 0. The first prediction is a hit and the second prediction is a miss, thus accuracy is 50%. Figure 9.2 plots the cross-entropy loss when these predictions come from probabilities ranging [0.5, 0.9]. One can observe that on the right side of the figure, log loss is large due to misses with high probability, even though the accuracy is 50% in all cases.

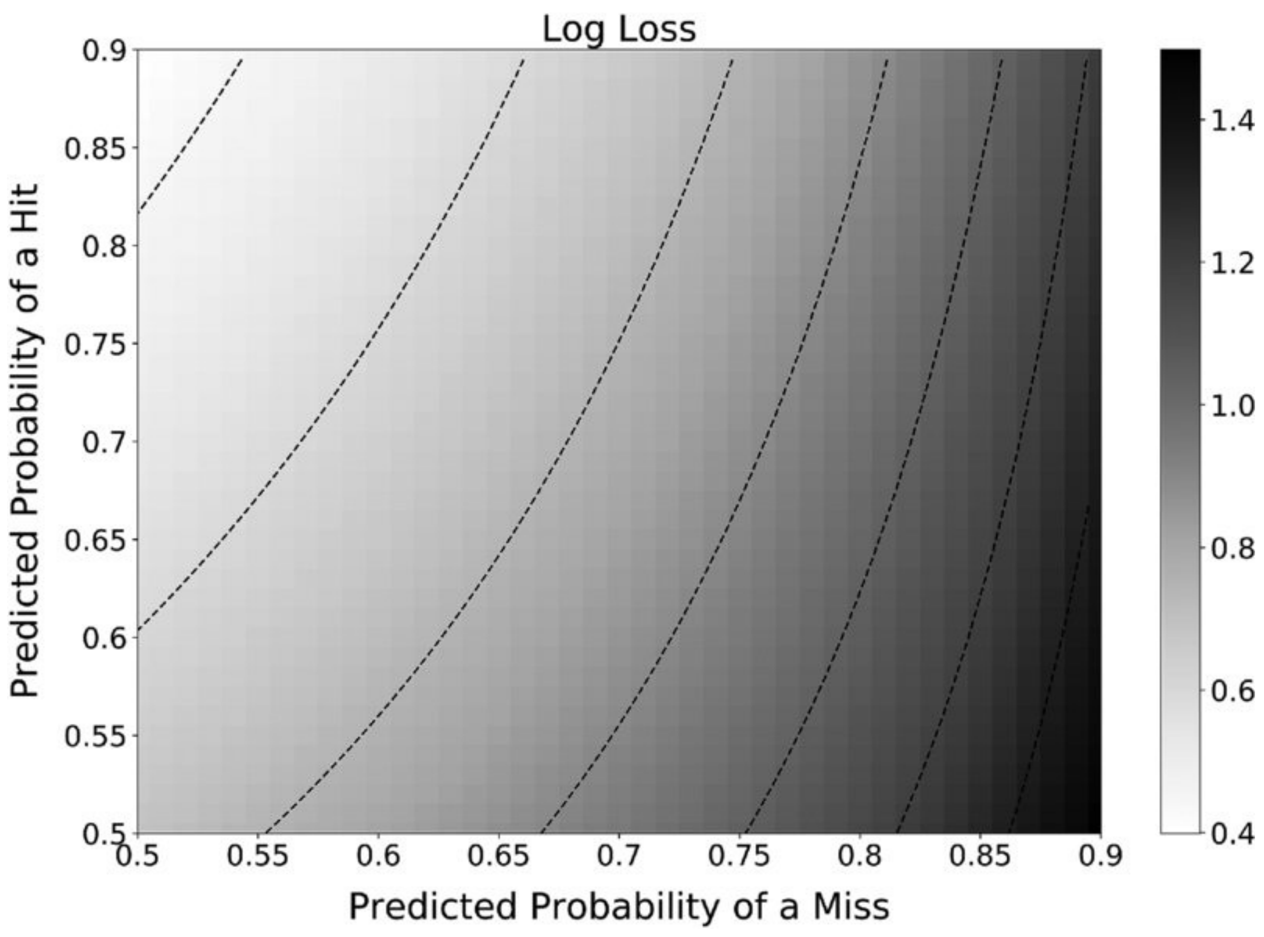

FIGURE 9.2 Log loss as a function of predicted probabilities of hit and miss

There is a second reason to prefer cross-entropy loss over accuracy. CV scores a classifier by applying sample weights (see Chapter 7, Section 7.5). As you may recall from Chapter 4, observation weights were determined as a function of the observation's absolute return. The implication is that sample weighted cross-entropy loss estimates the classifier's performance in terms of variables involved in a PnL (mark-to-market profit and losses) calculation: It uses the correct label for the side, probability for the position size, and sample weight for the observation's return/outcome. That is the right ML performance metric for hyper-parameter tuning of financial applications, not accuracy.

When we use log loss as a scoring statistic, we often prefer to change its sign, hence referring to "neg log loss." The reason for this change is cosmetic, driven by intuition: A high neg log loss value is preferred to a low neg log loss value, just as with accuracy. Keep in mind this sklearn bug when you use `neg_log_loss`: https://github.com/scikit-learn/scikit-learn/issues/9144. To circumvent this bug, you should use the `cvScore` function presented in Chapter

7.

EXERCISES

9.1 Using the function `getTestData` from Chapter 8, form a synthetic dataset of 10,000 observations with 10 features, where 5 are informative and 5 are noise.

a. Use `GridSearchCV` on 10-fold CV to find the `C`, `gamma` optimal hyper-parameters on a SVC with RBF kernel, where `param_grid = {'C':[1E-2,1E-1,1,10,100],'gamma':[1E-2,1E-1,1,10,100]}` and the scoring function is `neg_log_loss`.

b. How many nodes are there in the grid?

c. How many fits did it take to find the optimal solution?

d. How long did it take to find this solution?

e. How can you access the optimal result?

f. What is the CV score of the optimal parameter combination?

g. How can you pass sample weights to the SVC?

9.2 Using the same dataset from exercise 1,

a. Use `RandomizedSearchCV` on 10-fold CV to find the `C`, `gamma` optimal hyper-parameters on an SVC with RBF kernel, where `param_distributions = {‘C’:logUniform(a = 1E-2,b = 1E2),‘gamma’:logUniform(a = 1E-2,b = 1E2)},n_iter = 25` and `neg_log_loss` is the scoring function.

b. How long did it take to find this solution?

c. Is the optimal parameter combination similar to the one found in exercise 1?

d. What is the CV score of the optimal parameter combination? How does it compare to the CV score from exercise 1?

9.3 From exercise 1,

a. Compute the Sharpe ratio of the resulting in-sample forecasts, from point 1.a (see Chapter 14 for a definition of Sharpe ratio).

b. Repeat point 1.a, this time with `accuracy` as the scoring function.

Compute the in-sample forecasts derived from the hyper-tuned parameters.

c. What scoring method leads to higher (in-sample) Sharpe ratio?

9.4 From exercise 2,

a. Compute the Sharpe ratio of the resulting in-sample forecasts, from point 2.a.

b. Repeat point 2.a, this time with `accuracy` as the scoring function. Compute the in-sample forecasts derived from the hyper-tuned parameters.

c. What scoring method leads to higher (in-sample) Sharpe ratio?

9.5 Read the definition of log loss, $L[Y, P]$.

a. Why is the scoring function `neg_log_loss` defined as the negative log loss, $-L[Y, P]$?

b. What would be the outcome of maximizing the log loss, rather than the negative log loss?

9.6 Consider an investment strategy that sizes its bets equally, regardless of the forecast's confidence. In this case, what is a more appropriate scoring function for hyper-parameter tuning, accuracy or cross-entropy loss?

REFERENCES

Bergstra, J., R. Bardenet, Y. Bengio, and B. Kegl (2011): "Algorithms for hyper-parameter optimization." *Advances in Neural Information Processing Systems*, pp. 2546–2554.

Bergstra, J. and Y. Bengio (2012): "Random search for hyper-parameter optimization." *Journal of Machine Learning Research*, Vol. 13, pp. 281–305.

BIBLIOGRAPHY

Chapelle, O., V. Vapnik, O. Bousquet, and S. Mukherjee (2002): "Choosing multiple parameters for support vector machines." *Machine Learning*, Vol. 46, pp. 131–159.

Chuong, B., C. Foo, and A. Ng (2008): "Efficient multiple hyperparameter learning for log-linear models." *Advances in Neural Information Processing Systems*, Vol. 20. Available at http://ai.stanford.edu/~chuongdo/papers/learn_reg.pdf.

Gorissen, D., K. Crombecq, I. Couckuyt, P. Demeester, and T. Dhaene (2010): "A surrogate modeling and adaptive sampling toolbox for computer based design." *Journal of Machine Learning Research*, Vol. 11, pp. 2051–2055.

Hsu, C., C. Chang, and C. Lin (2010): "A practical guide to support vector classification." Technical report, National Taiwan University.

Hutter, F., H. Hoos, and K. Leyton-Brown (2011): "Sequential model-based optimization for general algorithm configuration." Proceedings of the 5th international conference on Learning and Intelligent Optimization, pp. 507–523.

Larsen, J., L. Hansen, C. Svarer, and M. Ohlsson (1996): "Design and regularization of neural networks: The optimal use of a validation set." Proceedings of the 1996 IEEE Signal Processing Society Workshop.

Maclaurin, D., D. Duvenaud, and R. Adams (2015): "Gradient-based hyperparameter optimization through reversible learning." Working paper. Available at https://arxiv.org/abs/1502.03492.

Martinez-Cantin, R. (2014): "BayesOpt: A Bayesian optimization library for nonlinear optimization, experimental design and bandits." *Journal of Machine Learning Research*, Vol. 15, pp. 3915–3919.

NOTES

[1] http://scikit-learn.org/stable/modules/metrics.html.

[2] http://scikit-learn.org/stable/auto_examples/svm/plot_rbf_parameters.html.

[3] http://scikit-learn.org/stable/modules/model_evaluation.html#log-loss.