UNIVERSITY OF COPENHAGEN

STATML EXAM

In a Galaxy Far, Far Away

Author:
Alexander WAHL-RASMUSSEN

KU Identity:
LGV740

April 3, 2014

1 Predicting the Specific Star Formation Rate

1.1 Linear Regression - Maximum Likelihood

For this section I chose the Maximum Likelihood approach to do linear regression. To solve the Maximum Likelihood, I first created a design matrix for the variables. I used a linear basis function of the form $\phi_i(x) = x_i$, and $\phi_0(x) = 1$, so that it looks like the following :

$$\Phi = \begin{pmatrix} 1 & \phi_1(x1) & \cdots & \phi_{M-1}(x1) \\ 1 & \phi_1(x2) & \cdots & \phi_{M-1}(x2) \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \phi_1(x_n) & \cdots & \phi_{M-1}(x_n) \end{pmatrix} \tag{1}$$

Where the first column marks the bias/offset parameter.
I then find the weight vector w_{ML} by applying the following formula:

$$w_{ML} = (\Phi^T\Phi)^{-1}\Phi^T t = \begin{pmatrix} -8.14943349 \\ -0.79400153 \\ -1.2229592 \\ -0.32858475 \\ -0.78633056 \end{pmatrix} \tag{2}$$

I then use the variables from (1) and (2) to predict the targeted values:

$$Y = w_{ML} * \Phi \tag{3}$$

The results for the train and test set can be seen in Table 1. Given that the Mean Squared Error can vary from 0 to infinity, with 0 being a perfect match and infinity being the worst possible match, the result we get seems quite good. It does indeed seem like we can approximate the logarithm of the sSFR with a linear regression. It is also worth noting that the error rate is almost equally low for both train and test set. This seems to indicate that both datasets have the same degree of noise and belong to the same domain, i.e. the data we learn on can be considered "typical" samples.

Table 1: Non-Linear Regression Results

Train Mean Squared Error	0.2748
Test Mean Squared Error	0.2752

1.2 Non-Linear Regression - Maximum A Posteriori

For the non-linear regression I chose the Maximum A Posteriori regression approach instead. I used the formula: $m_N = \beta S_N \Phi^T t$ for the mean and $S_N^{-1} = \alpha I + \beta \Phi^T \Phi$ for the covariance. I assigned the noise precision parameter β to 1, and set the range of α between 0 and 10, in increments of 0.5, where α is the prior. Adding the prior to the equation is equal to adding a regularization term for the Maximum Likelihood solution. It follows that one of the ways I can ensure good regularization is by controlling the value of α. This can be done via testing with e.g. cross-validation or having knowledge about the field or even both. For obvious reasons, I chose the former approach.

Additionally, I chose a polynomial regression with the degree varying from 2 to 5. I limited the degree as to prevent overfitting by creating a complex model with a high degree that only fits the presented data. The choice of maximum 5 is due to each data point only having 4 dimensions meaning any higher degree would lead to poor generalization.

To find the optimal parameters I then perform a gridsearch of the two parameters, degree & α, combined with a 5-fold cross-validation on the train set. The cross-validation then returns the parameter pair that yields the lowest average test error over the 5 folds. I then use those parameters to: train on the entire train set, test on both the train and test set, and the Mean Squared Error can be seen in Table 2

The results here are better than the ones we saw in the previous section. There might be several reasons for this. First of all, the data might simply fit better to a regression with a polynomial basis function. It might also indicate that the logarithmic sSFR trend is somewhat linear but with a high variance that is better explained with a polynomial of degree 3 with some shrinkage. More precisely, a 9.5 shrinkage! The latter seems to be the most probable explanation.

Finally, had I had another split or shuffled my data differently I would have had different parameters and therefore also a different result. I might have ended up with a lower This obviously implies that my result is not the 'ultimate' solution to the problem, but simply a well-fit approximation.

Table 2: Non-Linear Regression Results

α	9.5
Degree	3
Train Mean Squared Error	0.1698
Test Mean Squared Error	0.1911

2 Stars vs. Galaxies

2.1 Binary Classification with Support Vector Machines

For this section I used the the *SVC*[7] function, which is a SVM implementation that is based off the LIBSVM [2] library. I chose this library because I wanted a robust solution with an extensive documentation that would let me understand how everything is computed. The mathematical formulation of the learning problem SVC tries to satisfy is given in [7, 1.2.7.1. SVC], with the decision function being:

$$\operatorname{sgn}\left(\sum_{i=1}^{n} y_i \alpha_i K(x_i, x) + p\right) \tag{4}$$

where $y_i \alpha_i$ is the dual coefficients for the support vectors and p an individual constant term [1].

The kernel in this case is radial of the form $k(x, x') = \exp\left(-\gamma ||x - x'||^2\right)$, where the relevant hyperparamters are the influence term γ and the margin parameter C. For a high γ we reduce the influence radius of each data point while a low value will increase its influence. Likewise, a high value for C will enforce a hard margin where every train example will be classified correctly, while a small value will make the decision boundary more smooth with more errors on the train set. This

[1] For a more in-depth explanation of how the SVM works, see section 3.1 where I explain the classifier. In that example the SVM has a linear kernel, but the method is the same.

is also known as choosing a hard or soft margin. Not to mention, having a high value of C might not be wanted, because it can result in overfitting and therefore subpar generalization.

To achieve good results and an efficient solution, I first shuffled the data and then normalized it with zero mean and unit variance which reduced the training time. My approach was first to sample the $\gamma_Jaakkola$ heuristic from the entire dataset and then do a gridsearch for γ and C combined with a 5-fold cross-validation on the trainset while averaging over the 5-fold test errors. The best parameter pair is then pair that achieved the lowest average test error while cross-validating. Additionally, I chose to extend the range of γ to also include b^-4 and b^4 with b being Euler's number. The results for the entire train and test set can be seen in Table 3.

Table 3: RBF SVM Results

$\sigma_{jaakkola}$	1.018
$\gamma_{jaakkola}$	0.482
C_{Best}	20.086
γ_{Best}	0.024
Train Accuracy	99.47
Test Accuracy	99.5

2.2 Principal Components Analysis

For PCA I used the approach given in [1, pp. 561-563], which is the maximum variance method. I first sample the mean and variance from the dataset that corresponds to the galaxies. I then normalize the data with zero-mean and unit variance, not because it is requried for PCA or improves the performance, but because it improves the clustering performance in the next section. More on that there. I then resample the mean and the covariance matrix from the normalized distribution, where I compute the eigenvectors of the covariance matrix. The eigenvectors corresponds to the principal components of the distribution, where the eigenspectrum is plotted in Figure1. The low eigenvalues are due to normalization. I then project the data onto the two principal components with the highest eigenvalues by computing the dot product between the eigenvectors and the data points, where the eigenvectors are row vectors and the data points are vectors. The scatter plot of the dimensionality reduced data points can be seen in Figure2

2.3 Clustering

For this section i chose the K-means implementation from the Scikit-Learn library [4]. Although the Kmeans is fast by design, I still chose this library to ensure a robust and efficient solution, where the documentation is fulfilling. The K-means classifier is an unsupervised learning methods, meaning that it classifies by finding a hidden structure in the data that it tries to extract. It runs on unlabeled data which is why we have no exact way of measuring its error besides the judgement of our own interpretation - unlike every other classification method used in this exam. Specifically, the K-means clustering algorithm tries to sort the data into K clusters by minimizing the within cluster sum of square objective function. The mathematical formulation of this function, for a dataset X with n samples, is:

$$J(X, C) = \sum_{i=0}^{n} \min_{u_j \in C} (||x_j - \mu_i||^2) \tag{5}$$

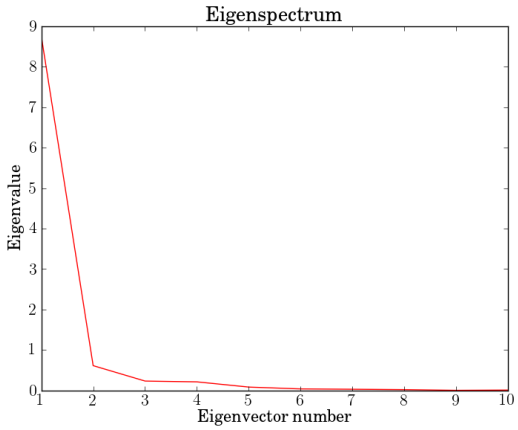

Figure 1: Eigenspectrum of the normalized SGData

As is seen in the equation, the function relies on the Euclidean distance, which performs best on normalized data. This is due to the distances between the data points with higher value ranges will be over-emphasised, i.e. the distance will seem bigger than they (relatively) are. Therefore I took the liberty of normalizing the data by zero mean and unit variance, which of course affects the location of the mean.

The algorithm itself runs in 3 steps. The first step is the initialization of the centroids, and in this case they are chosen at random. Step two is assigning every data point to its nearest mean with equation (5). Step three is measuring the difference between the old mean and the new mean. Step two and three are then repeated untill the difference value is less than the threshold of 0.0001 or the maximum number of iterations have been reached (300).

After the two centroids have been located, I project them with the same projection as in the previous section. The plot can be seen in Figure 3 and the Table 4.

As for the placement of the centroids; they seem quite intuitive. We almost see a white diagonal 'decision boundary' in the data around starting at (-4,5) and ending in (4,-3), which makes the centroid location even more plausible. As this is an unsupervised classification, I do not have any labels to measure the classification up again, so to truly answer the question it would require additional knowledge of galaxies, which I sadly do not have. According to the plot however, it seems meaningful.

2.4 Kernel Mean Classifier

For this part I chose to split the proof into two parts. Part one is the proof of the Euclidean distance, norm, in kernel space, while part two is the distance between a point and a mean in kernel space.

Claim 6 We define the Euclidean distance between two points x and z in vector space as:

$$||x - z||^2 = (x - z)^T(x - z) = x^T x + z^T z - 2z^T z$$

Where its kernel equivalent, for a positive definite kernel K, is:

$$||\phi(x) - \phi(z)||^2 = K(x, x') + K(z, z') - 2K(x, z) \tag{7}$$

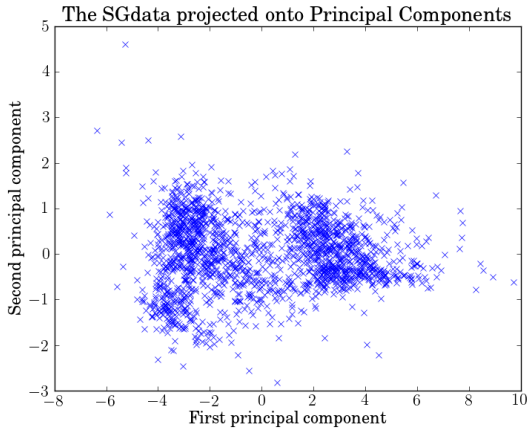

Figure 2: Projecting of the normalized data onto the 2 first Principal Components

Table 4: Centroid Coordinates

First Centroid	$\begin{pmatrix} 0.63444542 \\ 0.82603962 \\ 0.82280056 \\ 0.7893233 \\ 0.76804854 \\ 0.83342511 \\ 0.8068798 \\ 0.77489803 \\ 0.74606923 \end{pmatrix}$	Second Centroid	$\begin{pmatrix} -0.72644737 \\ -0.94582495 \\ -0.94211619 \\ -0.92809142 \\ -0.90378434 \\ -0.87942448 \\ -0.95428142 \\ -0.92388673 \\ -0.88726723 \\ -0.85425792 \end{pmatrix}$

Proof. Let ϕ be the transformation function for two given points x and z in vector space $\mathbb{R}^D$.

$$\begin{aligned} ||\phi(x) - \phi(z)||^2 &= (\phi(x) - \phi(z))^T(\phi(x) - \phi(z)) \\ &= (\phi(x)^T\phi(x)) + (\phi(z)^T\phi(z)) - 2\phi(x)^T\phi(z) \\ &= K(x, x') + K(z, z') - 2K(x, z) \qquad (K(x, x') = \phi(x)^T\phi(x)) \end{aligned}$$

□

Now that we have the definition of the Euclidean distance in kernel space we can define the kernel nearest mean classifier.

Claim 8 Given a training set with observations $X = \cup_{c \in C} X_c$ divided into a set C of classes X_c, the kernel nearest mean classifier is:

$$H(x) = \underset{c \in C}{\operatorname{argmin}}(K(x, x') + K(\mu(X_c), \mu(X_c)') - 2K(x, \mu(X_c))) \tag{9}$$

where $\mu(X_c)$ denotes the mean of all training points belonging to class c.

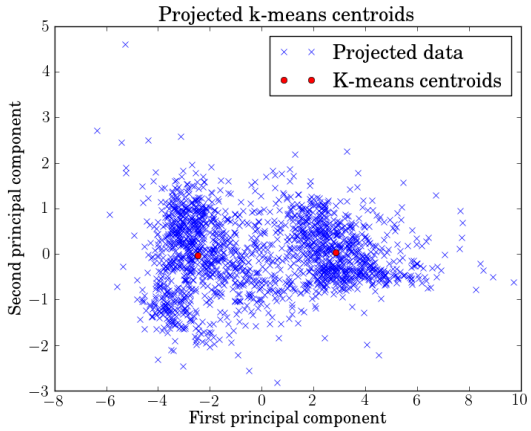

Figure 3: The centroids and data points projected onto the 2 Principal Components

Proof. Let $K(x, x')$ denote the kernel of (x, x') and $\mu(X_c)$ be the mean of all training points belonging to class c.

$$\begin{aligned} K(\mu(X_c), \mu(X_c)') &= K(\frac{1}{X}\sum_{i=1}^{X} x_i, \frac{1}{X}\sum_{j=1}^{X} x_j) \\ &= \frac{1}{X^2}(\sum_{i=1}^{X} x_i^T)(\sum_{j=1}^{X} x_j) \\ &= \frac{1}{X^2}\sum_{i=1}^{X}\sum_{j=1}^{X} K(x_i, x_j) \qquad \text{(Substituting as above)} \end{aligned}$$

The $K(x, (\mu(X_c))$ is then:

$$\begin{aligned} K(x, (\mu(X_c)) &= K(x, \frac{1}{X}\sum_{i=1}^{X} x_i) \\ &= \frac{1}{X}(\sum_{i=1}^{X} x_i^T)x \\ &= \frac{1}{X}\sum_{i=1}^{X} K(x_i, x) \qquad \text{(Yet another substitution!)} \end{aligned}$$

Knowing the $K(x, x')$ from beforehand and calculating the rest, we can now write the distance between a point and a class mean as:

$$\begin{aligned} ||\phi(x) - \phi\mu(X_c)||^2 &= K(x, x') + \frac{1}{X^2}\sum_{i=1}^{X}\sum_{j=1}^{X} K(x_i, x_j) - 2(\frac{1}{X}\sum_{i=1}^{X} K(x_i, x)) \\ &= K(x, x') + K(\mu(X_c), \mu(X_c)') - 2K(x, \mu(X_c)) \end{aligned}$$

And surrounding this equation with the $argmin_{c \in C}$ as to find us the nearest mean, we get (9):

$$H(x) = \underset{c \in C}{\operatorname{argmin}}(K(x, x') + K(\mu(X_c), \mu(X_c)') - 2K(x, \mu(X_c))) \tag{9}$$

□

3 Variable Stars

3.1 Multi-Class Classification

For this part I chose two classifiers with K-NearestNeighbor (KNN) as my non-linear, non-parametric classifier and Support Vector Machine with a linear kernel (SVM) as my linear, parametric classifier. Again I chose this library because of the extensive documentation. There are several reasons for why I chose these classifier and why I consider a SVM with a linear kernel as a linear classifier, as shall now become evident. However, let us start by examining how the algorithms classify. Imagine a case, where we only have 3 samples and 2 categories, in a n-dimensional space, and the first sample belongs to the first class while the second sample belongs to the other category. We then want to classify the third sample based on its features compared to the classified samples' features. This means we first train the specific algorithm on the two samples and then run the algorithm on the last sample.

The *(K-)NN* will then try to find the nearest neighbor by calculating the distance between the test sample and all other training data points. The nearest neighbor will then be the training data point that has the shortest distance to the test data point. The test data point is then classified as having the same class as its nearest neighbor, because their features are most alike. The amount of nearest neighbors to be considered can be increased to K - hence the name KNN, where the classification decision is based on majority voting between the neighbors. The distance metric itself can also vary, e.g. it can be Euclidean, Manhattan, Hamming etc., where the three also are known as direct, absolute and binary distances.

A *Linear SVM* will instead try to create a linear spatial decision boundary (in this case a straight line). The data is then divided onto each side of the decision boundary, making it a binary classifier. This resembles the approach of the Perceptron, but instead of trying to minimize the amount of errors, it creates the decision boundary at the maximum distance from the nearest support vectors. Intuitively this also means that the bigger the distance (or margin) is to the nearest support vectors, the lower amount of prediction errors should be on the seen data. But as stated earlier, having a hard margin may result in overfitting the train set, which we can control with the hyperparameter C. Unlike the Perceptron, the linear SVM can also classify non-linear distributions by utilizing the "kernel trick" and projecting the distribution into an unknown, but well-defined, high-dimensional space. In this space, the distribution will then become linearly separable and the decision boundary can be located. The kernel itself can also be changed to a non-linear function such as we have seen in chapter 2.1. I have simply run the linear SVM, which can arguably be described as a conceptual extension of the Perceptron.

The reason why I included the descriptions of the algorithms is quite simple: I believe that one of the biggest strengths of both classifiers is how they classify. That is, their approach to classification is very intuitive and quite simple[2]. It makes sense that the within class data should "look alike" and have a short distance to each other in the Euclidean space. It also makes perfect sense, that we should care about finding the *best* decision boundary, which is what SVM tries to do.

[2] Alright. Understanding and truly explaining SVM is not that simple, but its intution is both simple and straightforward

Furthermore, if I had used another linear classifier, such as the Perceptron, and had the dataset not been linearly separable in its current form, I would have had to make certain decisions. Would I want to stop the Perceptron at a certain iteration, which is a quite common approach[3], and if so, what iteration would I choose? Alternatively, I could have reduced the dimensions with PCA, or increased it with additional information such as the distance to the mean, to find the optimal dimension where the data is linearly separable. But why go through such hassle of finding the optimal dimensions (in addition to finding the optimal hyperparameters) when I can utilize the kernel trick to do the exact same thing without actually explicitly computing it.

So while it very well can be argued that a linear SVM is neither a linear classifier, since it does not linearly separate the data in the given space, nor a multi-class classifier, I believe that the steps you would otherwise take would result in almost the same approach.

3.1.1 Description of software used

For both classifiers I used implementations from the Scikit-Learn Python library[6].

The *KNeighborsClassifier* is a KNN implementation that also incorporates an underlying data-structuring algorithm such as a KDTree, Ball Tree or 'brute force'[5]. The brute force search is the 'vanilla' KNN where every data point is iterated over, while both KDTree and Ball Tree partitions the data into trees, where the KNN algorithm then iterates over the tree nodes instead. Any re-structuring of the data will of course affect the performance and the decision-making of the classifier, where the impact on the former is much higher than the latter. However, as I have a fairly limited knowledge of space-partitioning, I chose the beautiful setting of 'auto', where the function itself chooses an appropriate algorithm parameter as to achieve the most efficient solution. The only parameter I then have to supply is the amount of neighbors K. Additionally, the function .score(X,y) returns the accuracy instead of the 1-0 Loss and this I manually convert with $Loss = 1 - Accuracy$.

The *SVC*[7] is a SVM implementation that is based off the LIBSVM [2] library. It is the same implementation as the one in section 2.1 with the same decision function (4). As the SVM is a binary classifier it employs a one-by-one strategy for multiclass classification. This means that the function creates one classifier per class-pair, meaning the class with most votes among the subsets is chosen [1]. This is in contrast to the one-vs-all implementation, where each class is matched against the rest of the classes. I also chose the output to be deterministic and with the class weight set to 1, meaning each class has the same weight. Furthermore, as with KNN I convert the accuracy a 0-1 loss.

3.1.2 Results

To achieve the best results I first normalized the data by making it zero-mean with unit variance as I explained in the previous sections. I chose to normalize my data for two reasons. Since KNN matches by minimum distances it inherently assumes that all our data has the same range of values, which means that distances between the data points with higher value ranges will be over-emphasised. This can be avoided by normalizing. SVM does not explicitly suffer from this problem since it only considers the support vectors and not the entire dataset. But due to its bad scaling behavior it improves the training time to reduce the value range and it forces both classifiers to perform on equal footing.

I then perform a 5-fold cross-validation on the train set to find the optimal parameters K and C. K is a list of integers between 1 & 15 and C is a list numbers from 0.005 to 1 with step size 0.005.

[3] Even the Scikit-Learn has a low max iteration declared as a default!

[1] This means that for a case of n classes there will be a classifier for each pair: 0 vs 1, 0 vs 2, .. 0 vs n, 1 vs 2, .. n-1 vs n

Initially I also tested $C_i > 1$ but that did not increase the test result, which leaves us with a soft margin (C¡1). I then return the parameters that gave the lowest average test loss. The results from the *entire* train and test set can be seen below. To see the results of the cross-validation you will have to run the code.

Table 5: Multi-Class Classification Results

Classifier	Hyperparameter	Train Error	Test Error
KNN	K = 8	0.280	0.341
Linear SVM	C = 0.06	0.147	0.265

At first glance the results do not seem very impressive. But, if we consider that each data point has 61 features and can belong to one of 25 different classes *and* that there's only 771 training and equally many test samples, the results do not seem that shabby. Usually you would want a 80/20 or even 90/10 split between train and test size as to achieve good results. It is therefore not too surprising that the results do not reach a lower error rate. That is even without considering the sparsity of the dataset compared to the dimensionality of each data point. And that class 2 only has 1 data point! This means KNN will never classify it correctly. The sparsity will also affect the cross-validation negatively, since some classes might be absent from one or more folds, which would lead to a poorer performance. The relatively high value of K and low value of C also indicates that the data is quite noisy, which again also reduces the performance of the cross-validation. However, I do not think splitting the train set into a train/validate test set would prove any better due to the sparsity of the data.

It is also worth noting that my knowledge of the relationship between machine learning and star-detection is quite limited. But I know that it is at least much better than a random cointoss! As to improve on the results, it might be beneficial to reduce the dimensionality with e.g. PCA to diminish the amount of noisy dimensions and then employ the kernel trick.

3.2 Overfitting

Out of the different measures of overfitting, I chose data set selection, parameter tweak overfitting and old datasets from John Lanford's blog [3].

Dataset selection I believe dataset selection to be the most prevalent method of overfitting. Not because we are not aware of how we select data sets, but exactly because we *are* aware of the selection. Selection inherently implies choosing something over something else which results in bias. We simply cannot avoid *some* bias in our dataset selection, making any machine learning method overfit to the specific data set's structure instead of the natural's.

I am aware that Langford does not outright say this himself, but I believe it to be an important aspect of data set selection. I also find his remedies good, but they ultimately do not solve the issue of data set selection, but they do diminish the amount of overfitting, which is what we should strive for. Selecting a truly unbiased data set is therefore closer to a quest for Don Quixote than for a data scientist. On the other hand, diminishing the amount of selection bias, and thereby overfitting to the data sets, should be at the core of machine learning methodology. Naturally, all of the above applies to the the dataset given to us in the Variable Stars assignment as it has been selected for a specific purpose.

Parameter tweak overfitting is a temptation every data scientist has faced at least once. Per experiment. Per test run. Because, would it not be nice to get that 1 percentage better results that the rest? However, this will undeniably result in overfitting to the test set as opposed to the train set. But this is just as bad. Why? Because we do not know anything about how good our model is at generalizing, which is why we have a test set in the first place. In our case, we want our model to learn the variable stars' structure, not learn it by heart. That is, we want to be able to acquire knowledge the subject and have a model we can depend on in the future. The same goes for choosing a highly complex model. The likelihood of the complex model describing anything else but the exact dataset is not high.

Old datasets I do not know much about stars. But with my fairly limited knowledge of light's passage through space, I at least know this: The intensity we observe at any given time is already old at the time we observe it. Why? Because stars are quite far away and light travels at a certain speed. So the most important question is: does it matter that the dataset itself is old and has old data? In my opinion, it depends on the consistency in the data we observe. As Langford says, as datasets get released and become older, we can develop algorithms that perform extremely well on those specific datasets. They so to speak overfit to the data set, which we also discussed above, and they *might* not perform well on future datasets. If the data is very consistent and dataset has followed a sturdy selection methodology, we should theoretically expect any classifier trained on the old dataset to perform equally well on new datasets.

But things are rarely consistent. Just try to train a classifier with 'Wall Street Journal'-English and then classify texts in "Twitter-speak". The results will not be pretty. If they are not consistent we end up in the same situation as with the data set selection, where our classifiers will overfit to a dataset instead of natural structures. As Langford says, we can prevent this to a certain extent by weighting newer dataset higher or not make test examples publicly available. But this does not prevent it entirely. It is therefore important to constantly update the training datasets so that our approximations become more approximate instead of less approximate.

References

[1] BISHOP, C. M. *Pattern Recognition and Machine Learning*. Springer, 2006.

[2] CHANG, C.-C., AND LIN, C.-J. LIBSVM. `http://www.csie.ntu.edu.tw/~cjlin/libsvm/`, March 2014.

[3] LANGORD, J. Clever methods of overfitting. http://hunch.net/?p=22.

[4] SCIKIT-LEARN. Clustering. http://scikit-learn.org/stable/modules/clustering.html.

[5] SCIKIT-LEARN. KNeighborsClassifier. `http://scikit-learn.org/stable/modules/neighbors.html`, March 2014.

[6] SCIKIT-LEARN. Machine Learning in Python. `http://scikit-learn.org`, March 2014.

[7] SCIKIT-LEARN. Support Vector Machines. `http://scikit-learn.org/stable/modules/svm.html`, March 2014.