Keep It Simple: Is Deep Learning Good for Linguistic Smell Detection?

Sarah Fakhoury, Venera Arnaoudova
School of Electrical Engineering and Computer Science
Washington State University
Pullman, USA
{sarah.fakhoury, venera.arnaoudova}@wsu.edu

Cedric Noiseux, Foutse Khomh, Giuliano Antoniol
Department of Computer and Software Engineering
Polytechnique Montréal
Montréal, Canada
{cedric.noiseux, foutse.khomh, giuliano.antoniol}@polymtl.ca

***Abstract*—Deep neural networks is a popular technique that has been applied successfully to domains such as image processing, sentiment analysis, speech recognition, and computational linguistic. Deep neural networks are machine learning algorithms that, in general, require a labeled set of positive and negative examples that are used to tune hyper-parameters and adjust model coefficients to learn a prediction function. Recently, deep neural networks have also been successfully applied to certain software engineering problem domains (e.g., bug prediction), however, results are shown to be outperformed by traditional machine learning approaches in other domains (e.g., recovering links between entries in a discussion forum).**

In this paper, we report our experience in building an automatic Linguistic Antipattern Detector (LAPD) using deep neural networks. We manually build and validate an oracle of around 1,700 instances and create binary classification models using traditional machine learning approaches and Convolutional Neural Networks. Our experience is that, considering the size of the oracle, the available hardware and software, as well as the theory to interpret results, deep neural networks are outperformed by traditional machine learning algorithms in terms of all evaluation metrics we used and resources (time and memory).

Therefore, although deep learning is reported to produce results comparable and even superior to human experts for certain complex tasks, it does not seem to be a good fit for simple classification tasks like smell detection. Researchers and practitioners should be careful when selecting machine learning models for the problem at hand.

***Index Terms*—Antipattern Detection, Machine Learning, Convolutional Neural Networks.**

I. Introduction

Deep neural networks, and more generally, deep learning approaches have proven useful in many fields and applications. They are able to extract, find and summarize complex patterns in large data sets. The hype is that they substantially simplify the development of sophisticated recommendation or classification systems.

Recently, deep neural networks have also been applied to certain software engineering problems with some success. For example, they were able to improve upon the best known results in traceability recovery between software artifacts [1], and exciting results have been achieved in semantic clone detection [2]. However, not all problems are equal. There are problem instances, such as bug prediction or identifier completion where training material is abundant and freely available. Other problems may require extensive manual effort and validation to build a labeled dataset. Regardless of the problem at hand, one should always apply *"lex parsimoniae"* also known as *Occam's razor*. Between two competing theories or models giving the same results, the simplest one to understand and interpret should be preferred. Indeed, one has to balance different factors. On the one hand, the ability to capture, summarize, and model information and dependencies makes deep neural networks extremely appealing; on the other hand, they require complex hyper-parameter and architecture tuning, sizable labeled datasets for training, and quite powerful hardware architectures and resources.

In this paper, we report our experience of building an automatic Linguistic Antipattern Detector (LAPD) using traditional machine learning approaches and deep neural networks, namely, Convolutional Neural Network (CNN). At the high level our research goal can be stated as: *Do traditional machine learning classifiers outperform CNN for the task of linguistic smell detection?* In other words, are Fu and Menzies results [3] reproducible? To achieve our goal, we apply the same deep learning architecture used by Fu and Menzies [3], namely, CNN, and similarly, we compare against traditional machine learning algorithms. However, we applied a structured methodology to define CNN hyper-parameter and architecture inspired by Zhang and Wallace [4], [5]. In addition, we have manually built and validated an oracle of around 1,700 instances (half of which contain linguistic antipatterns) and created binary classification models using Random Forest, SVM, J48, Naïve Bayes, and CNN.

Our results confirm the findings and recommendations of the recent work by Fu and Menzies [3]; on a completely different task we find that traditional approaches outperform CNN in terms of all evaluation metrics that we used, as well as, in terms of time and memory resources. Considering the size of the oracle, the available hardware and software, as well as theory to interpret results, deep neural networks do not pass the Occam's razor test. They are cumbersome to define (the software configuration hugely depends on the hardware platform where code is executed), they are slow, resource hungry, and almost impossible to interpret. On the contrary, traditional machine learning algorithms run much faster, require far less tuning, can run on a standard laptop, and ultimately achieve from similar to better results depending on the tuning used

978-1-5386-4969-5/18/$31.00 © 2018 IEEE

SANER 2018, Campobasso, Italy
RENE Track

to configure them. Furthermore, the complexity of the deep learning approach hinders the ability of researchers to test the stability of their conclusions and the potential for replication and improvement of the results by others.

This paper makes the following contributions:

- We provide evidence that when correctly tuned, traditional machine learning classifiers can substantially outperform deep learning approaches in the context of smell detection
- We successfully replicate the results of Fu and Menzies and provide a more methodological exploration of deep learning vs traditional machine learning classifiers within the problem space.
- We provide an oracle of 1,753 instances of source code elements and the LA type that they contain, if any, to support future research on LAs. A curated replication package containing the manually built oracle is available online[1]

Paper Organization. The rest of the paper is organized as follows. In Section II, we present background information about LAs and ML classifiers. We survey the related work in Section III. In Section IV, we present our analysis approach. We answer the research questions in Section V and discuss the threats to validity of our study in Section VI. We conclude the paper in Section VII.

II. Background

In this Section, we provide a brief background on Linguistic Antipatterns (Section II-A), on the Machine Learning algorithms that we used (Section II-B), and on CNNs (Section II-C).

A. Linguistic Antipatterns (LAs)

Linguistic Antipatterns (LAs), are recurring poor practices in the naming, documentation, and choice of identifiers in the implementation of program entities. These LAs are reported to possibly impair program understanding [6]. In the remainder of this section, we list the LAs and their definition as described by Arnaoudova et al. [6].

A.1 *"Get" - more than an accessor*: A getter that performs actions other than returning the corresponding attribute without documenting it.

A.2 *"Is" returns more than a Boolean*: Method name is a predicate, whereas the return type is not Boolean but a more complex type allowing a wider range of values.

A.3 *"Set" method returns*: A set method having a return type different than `void` and not documenting the return type/values with an appropriate comment.

A.4 *Expecting but not getting a single instance*: Method name indicates that a single object is returned but the return type is a collection.

B.1 *Not implemented condition*: The method' comments suggest a conditional behavior that is not implemented in the code. When the implementation is default this should be documented.

B.2 *Validation method does not confirm*: A validation method that neither provides a return value informing whether the validation was successful, nor it documents how to proceed to understand.

B.3 *"Get" method does not return*: The name suggests that the method returns something (e.g., name starts with "get" or "return") but the return type is `void`. The documentation should explain where the resulting data is stored and how to obtain it.

B.4 *Not answered question*: The method name is in the form of predicate, whereas nothing is returned.

B.5 *Transform method does not return*: The method name suggests the transformation of an object, however there is no return value and it is not clear from the documentation where the result is stored.

B.6 *Expecting but not getting a collection*: The method name suggests that a collection should be returned, but a single object or nothing is returned.

B.7 *Get method does not return corresponding attribute*: A get method does not return the attribute suggested by its name.

C.1 *Method name and return type are opposite*: The intent of the method suggested by its name is in contradiction with what it returns.

C.2 *Method signature and comment are opposite*: The documentation of a method is in contradiction with its declaration.

D.1 *Says one but contains many*: An attribute name suggests a single instance, while its type suggests that the attribute stores a collection of objects.

D.2 *Name suggests Boolean but type does not*: The name of an attribute suggests that its value is true or false, but its declaring type is not Boolean.

E.1 *Says many but contains one*: Attribute name suggests multiple objects, but its type suggests a single one.

F.1 *Attribute name and type are opposite*: The name of an attribute is in contradiction with its type as they contain antonyms.

F.2 *Attribute signature and comment are opposite*: Attribute declaration is in contradiction with its documentation.

B. Machine Learning (ML) Models

We select five different machine learning approaches belonging to several different categories: rule learners, decision trees, Support Vector Machines, and Bayesian Networks. Recent research has shown that these algorithms perform well when predicting fault-prone code [7]–[9]. We use the implementations provided through Weka [10] and then tune the classifiers by hand using various thresholds. We then use a more sophisticated approach, based on bayesian optimization, to find the most suitable model, hyperparameters and features for the problem. The following is a list of the selected algorithms, their descriptions and thresholds used during manual tuning:

[1] https://github.com/Smfakhoury/SANER-2018-KeepItSimple-

1) Random Forest (RF): RF [11] averages the predictions of a number of tree predictors where each tree is fully grown and is based on independently sampled values. A large number of trees avoids overfitting. Random Forest is known to be robust to noise and to correlated variables. We use a number of trees of 500 as a starting point, which has shown good results in previous works [12]. We tune the parameters for the number of trees varying from 500 to 1000 and for the features explored at each branch from the default value: $(log_2(\#predictors)+1)$ to 20% of the total number of features.

2) J48: J48 is an implementation of the C4.5 decision tree. This algorithm produces human understandable rules for the classification of new instances. The implementation provided through Weka offers three different approaches to compute the decision trees, based on the type of the pruning techniques: pruned, unpruned, and reduced error pruning. We tune the parameter for the minimum number of instances at each leaf from 1 to 8.

3) Support Vector Machine (SVM): SVM is a machine learning technique that tries to maximize the margin of the hyperplane separating different classifications. Some of the advantages of SVM include the possibility to model linear and non-linear relations between variables and its robustness to outliers. We used the Support Vector Machine model provided by LibSVM in Weka with the Radial Basis Function (RBF) and Polynomial kernels as they have shown good performance in previous work [13]. We tune the parameters for the degree of the kernel (1 to 4), the gamma value (0 to 1) and the cost (1 to 50) based on recommendations from [3].

4) Sequential Minimal Optimization (SMO): SMO is an implementation of John Platt's sequential minimal optimization algorithm to train a support vector classifier. We use RBF kernel, Polynomial kernel, and the Pearson VII function-based universal kernel (PUK) [14] in combination with this classifier. We tune the exponent parameter of the classifier from 1.0 to 4.0.

5) Naive Bayes: Naive Bayes is the simplest probabilistic classifier based on applying Bayes' theorem. It makes strong assumptions on the input: the features are considered conditionally independent among each other. We explore the performance of the classifier using kernel estimator and supervised discretization.

C. Convolutional Neural Networks (CNNs)

Several deep learning architectures are available, however, due to the textual nature of our data (i.e., sequences of comment and source code tokens) we decided to explore CNN.

The idea of using a CNN was inspired by a paper written by Kim [15]. He used a CNN for sentence-level classification tasks and showed that one can achieve excellent performance results with little parameter calibration.

Typically, CNNs were employed for image classification [16]. A CNN can be thought of as a family of convolutions filters, as windows sliding across a whole matrix. These windows are acting like filters extracting (summarizing) information. For images, this matrix contains pixels, for sentences, it contains word representations (e.g., vectors of word embeddings–Google word2vec). Each sentence is represented as a sequence of words and each word as a vector. Finally, since sentences may have different length, matrices are zero padded to the maximum sentence length.

To implement a CNN, one has to add several convolution layers; each of these layers have a specific task and acts as a different filter. Convolution layers are then followed by a pooling layer. Each filter generates a feature vector of length depending on the sentence length and filter size. Different sentence lengths and filter sizes will generate feature vectors of different size. This is not practical and it is cumbersome to handle. A common strategy is to apply a max-pooling layer which extracts the largest value from each feature. This is to say, if we apply 16 filters of size two and three, no matter the sentence length, after the pooling layer we will always have 16 values for the two size windows and 16 values for the filter of size three. In other words, each sentence will be mapped to a set of 32 feature values. The output of the pooling layer is then fed into a fully connected layer to the output categories.

We experimented with one convolution layer, one pooling layer, and a fully connected output later; this is to say, we replicated the CNN structure studied and presented by Kim [15] and by Zhang and Wallace [4], [5]; this allowed us to reuse insights and perform the same methodological steps to identify promising configurations (i.e., number of filters and filter sizes). Also, we observed that words encoding specifically tied to the domain and task (i.e., non-static word2vect CNN encoding in the work by Zhang and Wallace [4]) perform comparably or equally well as more complex encoding (e.g., encodings obtained using GloVe and static word2vect) [4], [5]. Therefore, all CNN results reported in the paper are computed with customized word embeddings referred to as dynamic word2vect by Kim [15], Zhang and Wallace [4], [5].

III. Related Work

The most relevant works are application of deep learning to sentence classification [4], [5], [15] and a very recent paper in software engineering [3].

We were inspired to use CNN by the 2014 Kim's work [15]. Kim compared on seven established data sets (e.g., MR, TREC, MPQA) four CNN variants (e.g., static, multi-channel) in term of accuracy with previously obtained results in 10-fold cross validation experiments. Overall, he considered 14 previous papers that using a variety of classifiers (including recursive neural networks, recursive auto-encoders, Naive Bayes SVM, SVM manually tuned) have published classification results on the same data sets. Interestingly, Kim found that CNN performed (on average) equally or even better than most classifiers but CNN was not always the best classifier. In particular, on the TREC dataset the best classifier was a manually tuned SVM. This corroborated us to experiment with CNN as LAs are essentially linguistic structures but being unsure of CNN performance we also decided to contrast CNN results with other classifies as done by Kim. Kim finding

that CNN were not always the best classifier, was recently confirmed on a software engineering problem and dataset by Fu and Menzies [3]; they found that CNN was not performing better than a carefully tuned SVM.

Also the work of Zhang and Wallace [4], [5] has been an inspiration for us. We define the same CNN architecture and follow the same methodology to tune the hyper-parameters. This is important as it gives us a reference point and a way to select various parameters namely: type of embeddings, number of features and, windows size, gradient optimization strategy and activation function. Last but not least, commonalities exist between this work and the work by Fu and Menzies [3]. Indeed, we have similar findings on a different task: CNN are not necessarily better than traditional machine learning approaches. However, Fu and Menzies use SVM as the traditional machine learning algorithm whereas we use Random Forest, SVM, J48, and Naïve Bayes. In addition, Fu and Menzies use Differential Evolution to tune the parameters of SVM whereas we use basic parameter tuning for all traditional machine learning algorithms and we use more sophisticated tuning, based on Bayesian optimization, to find the best model and features. We show that standard tuning approaches of traditional machine learning algorithms lead to results similar to results obtained with CNN while with sophisticated tuning of traditional machine learning algorithms significantly outperforms results obtained with CNN.

IV. Methodology

The *goal* of this study is to determine if deep learning is well suited for detecting linguistic smells, and whether traditional machine learning classifiers can outperform CNN. The *perspective* is that of researchers interested in developing and evaluating approaches able to recognize LAs, and practitioners interested in removing LAs from their software systems. The study aims at addressing the following research questions:

RQ1: *Can we use CNN for linguistic smell detection?* We explore the performance of deep learning algorithms, particularly CNNs, to detect linguistic smells in source code.

RQ2: *Do traditional ML classifiers outperform CNN for linguistic smell detection?* We investigate whether traditional, less resource-consuming classifiers can achieve close, or even better performance than deep learning alternatives in terms of precision, recall, F-measure, ROC, and MCC.

RQ3: *Are traditional ML classifiers faster than CNNs?* We compare the speed in which traditional machine learning classifiers and CNN can learn a prediction model.

Figure 1 depicts an overview of our approach. The input is Java source code on which we run the rule-based LAPD to detect LAs. To create an oracle, we randomly sample and manually validate instances of methods and attributes categorized as LAs and non-LAs by LAPD. We use the Java source code as well to extract features for the ML classifiers as follows: we collect and preprocess source code entities and tag them with part of speech, grammatical dependencies, semantic relations, and srcML tags. As part of **RQ1**, we use the preprocessed source code as input to the CNN. For **RQ2**, the various ML algorithms are explored using features extracted from the preprocessed source code. Finally, we evaluate the performance of the approaches in **RQ1** and **RQ2** using the oracle and the evaluation metrics described in Section IV-F.

The rest of this section is organized as follows: Section IV-A provides details about the systems used in this study while Section IV-B describes the process that we followed to create the oracle. Section IV-C describes the feature extraction and data preprocessing. Section IV-D describes how we tune traditional machine learning classifiers. Section IV-F describes how the different approaches are evaluated. Section IV-G details the analysis methods we used to answer our research questions.

A. Dataset

For the purpose of this study we select 13 open-source Java systems. Table I summarizes the characteristics of the dataset, namely the system releases analyzed, their size in terms of number of lines of code (KLOC), and the number of detected LAs by the rule-based LAPD. System size varies from small (Junit) to medium (Apache Lucene). Selected systems belong to different application domains to avoid dependencies of results on a specific domain or ecosystem.

TABLE I
Subject systems

System Release	Size (KLOC)	# of Detected LAs
Apache Lucene 6.6.1	1473	488
Apache Beam	246	666
Eclipse 1.0	774	1489
ArgoUML 0.34	391	443
Apache Tomcat 9.0.0	539	1246
Rhino 1.7.7.1	129	326
Apache maven 3.3.6	129	787
Apache spark 2.0.0	141	338
Apache Kafka 0.11.0.1	234	706
Cocoon 2.2.0	61	204
OpenCv 3.3.0	23	2262
JavaCv 1.1	22	254
JUnit 4	43	153

B. Oracle

To build the oracle, we run LAPD[2] on each project and use the generated log files to randomly select a subset of LAs per project. Each LA was then manually validated by two evaluators. The kappa agreement between the evaluators was calculated after validation of instances in intervals of 50. The kappa values range from 0.80 to 0.88 which indicates substantial to almost perfect agreement [17]. Instances on which the evaluators disagreed would be discussed and a conclusion would try to be reached, the instance was discarded otherwise.

In the case of a doubtful instance, where the presence of the LA was not certain, the instance was also discarded. We followed a similar procedure to randomly select and manually

[2]http://www.veneraarnaoudova.com/linguistic-anti-pattern-detector-lapd/

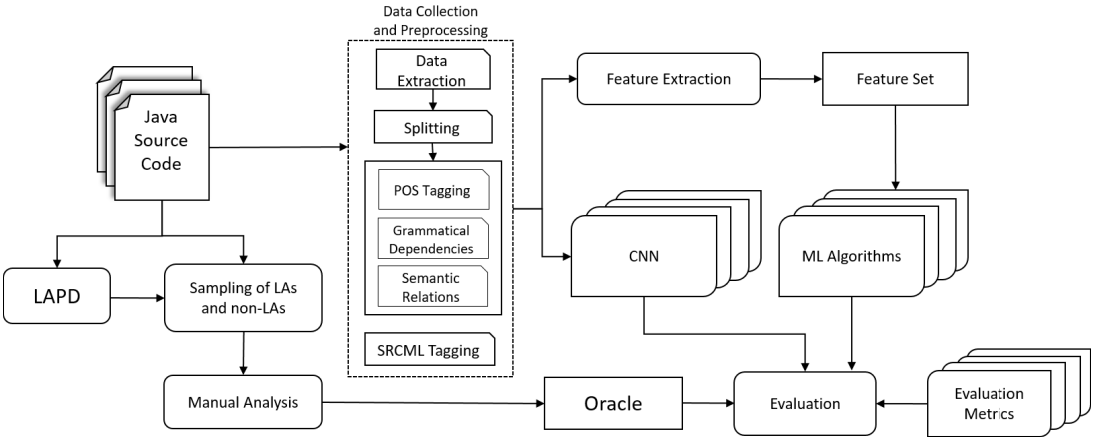

Fig. 1. Overview of the approach.

validate instances that LAPD reported as not being LAs. We validated a total of 1753 instances in our oracle, derived from thirteen open-source projects. Out of the 1753 instances of code entities (methods and attributes), 808 instances contain LAs of 18 different types and 945 instances are validated as not containing any LA. Instances in the oracle are labeled with the type of LA. However, for the purpose of this study, we only build binary classifiers of the instances predicting whether a source code entity contains a LA or not, thus combining all types of LAs into one group.

C. Data Collection and Preprocessing

We use several preprocessing techniques to remove noise from the dataset and facilitate prediction. All special non alphabetic characters in the code are removed, retaining only the lexical information relevant to the identification of LAs. LAPD uses parts of speech, semantic relations, and grammatic dependencies to lexically characterize source code as part of its rule-based detection. We use the type of information extracted by the LAPD as a starting point for our dataset and then add supplemental information to aid smell detection. A detailed explanation of what kind of information is extracted and how it is processed follows.

1) Data Extraction: When extracting instances from the source code, we extract the entire source code element. Thus, for source code entities involving methods, we extract the method signature, method body, and the comment of the method (if applicable). For source code entities involving attributes we extract the entire attribute declaration statement and any accompanying comments.

2) Splitting: Splitting involves identifying individual terms in a composed identifier. For example, the identifier `getMessage` must be split into its two composing terms: `get` and `Message`. The resulting terms can be dictionary words, abbreviations, or acronyms. In the context of Java source code, splitting identifiers using camelCase and underscore heuristics is sufficient [18].

3) Part Of Speech Tagging: After identifiers have been split into terms, we build a sentence out of the terms and perform part of speech analysis to identify the roles and relations of specific terms within the sentence. We use Stanford CoreNLP [19]—a set of tools that facilitates POS analysis and identification of grammatical dependencies between words. The Stanford CoreNLP classifies terms using the Penn Treebank Tagset [20], and can distinguish between nouns, verbs, adjectives, and adverbs, as well as their different forms, e.g., plural noun, verb past participle, etc. For example, the terms of the identifier `getMessage` are tagged as verb (`VB`) and singular noun (`NN`), respectively.

4) Grammatical Dependencies: Grammatical dependencies are direct grammatical relations between terms in a sentence. For example, the identifier `setUserId` will first be transformed into the sentence `set user id`. Then tagging the sentence with the universal dependency scheme we obtain `root(ROOT,set)` where `set` is the root of the sentence, `compound(id,user)` a compound relation between `id` and `user`, and `dobj(set,id)` a direct object relation between `set` and `id`. For our dataset, we extract grammatical dependencies on every comment and for each individual identifier.

5) Semantic Relations: To find semantic relations between terms, we use WordNet, a general English ontology. Words in WordNet are organized based on their relations. Synonyms are grouped into unordered sets, called synsets, which in turn are related using semantic and lexical relations. Thus, using WordNet, we are able to identify semantic relations among the terms in the source code and comments. Precisely, we extract synonym and antonym relations.

6) srcML Tagging: srcML is a tool developed by Collard et al. [21] to parse source code into an equivalent XML representation. srcML wraps the source code with information from the Abstract Syntax Tree (AST) and generates tags for various AST elements. We use this information to tag the extracted elements with the AST elements it contains. This is to supply the learner with potentially useful information, for example, to differentiate between a method comment and an identifier name.

D. Traditional Machine Learning Classifier Tuning

Fu and Menzies use differential evolution hyper parameter optimization to tune SVM. Similar work in software engineering [22] [23] has shown that random search algorithms, such as the differential evolution algorithm, perform better than iterative algorithms, such as grid search, in term of speed and efficiency. However, Bayesian optimization methods have been shown to rival and even outperform alternative approaches for hyperparameter optimization, including random search algorithms [24] [25] [26].

We use Auto-Weka [27], a system designed to automatically perform attribute selection, search for the optimal learning algorithm, and hyperparameter settings to maximize performance using state-of-the-art Bayesian optimization techniques. Auto-Weka allows the user to configure time and space constraints for the tuning process and returns the best performing classifier found within the set constraints, optimizing for the evaluation metric of choice. Similar to Fu and Menzies, we use F1-score as the tuning goal for Auto-Weka, which means that when tuning parameters, Auto-Weka finds a combination of parameters that balance precision and recall to maximize the F1-score. To evaluate the tuned model consistently with the manually tuned classifiers and the CNN, we configure Auto-Weka to use leave-one-out cross validation.

E. CNN Tuning

Experiments have been executed by adapting existing CNN project available on GitHub[3]. The code was modified to pass as parameter the name of the word embedding dictionary and to store results. This was done to avoid recomputing the embeddings each time and save resources.

Words embedding have been computed by means of Google word2vec. We used the efficient C implementation available on GitHub[4]. In computing the embedding we reused the suggested standard parameters, i.e., trained via bag of words model; a window of five; number of negative examples set to five; threshold for downsampling the frequent words $1e^{-4}$; and no hierarchical softmax. We experimented with various embedding sizes and we retained 150 as a compromise to keep training process memory manageable (i.e., below 20 Gb).

1) CNN Configuration: To tune the model we selected about one third of all datapoints i.e., about 500 instances and run several experiments to define CNN architecture. Similar to Zhang and Wallace [4], [5] we started computing several simple models (e.g., using one filter type, multiple filter types of the same size, mixed filter sizes). Due to space reason we succinctly describe results here.

We experimented with sizes from one to seven since in previous studies larger windows improved accuracy marginally (less than 2%) only on one data set (CR [4]). We made the number of filters to vary: 16, 32, 64, and 128 filters, again the choice was motivated by previous results (a larger number of features only marginally affects accuracy [4].

[3]CNN Text Project: https://github.com/dennybritz/cnn-text-classification-tf

[4]word2vec: https://github.com/dav/word2vec

Results of single filter type show that the best possible size is between two and five with no clear winner in the filter numbers. We then run a second set of experiments. This time for each candidate filter length (say four) we considered multiple windows of the same size. The idea is to attempt modeling relations between different regions of the same length. For example, for a 16 filters if we have a combination five, five it means we have two times 16 filters, where in both families each filter has size five. In this experiment we considered combinations from two to four regions.

TABLE II
CNN 128 FILTERS MULTIPLE REGIONS CLASSIFIER PERFORMANCE

SIZE	REC.	PREC.	SPECIFICITY	F1	ACC.
2,2	49.02	66.67	92.09	56.50	81.58
2,2,2	46.08	65.28	92.09	54.02	80.86
2,2,2,2	48.04	65.33	91.77	55.37	81.10
3,3	45.10	69.70	93.67	54.76	81.82
3,3,3	46.08	73.44	94.62	56.63	82.78
3,3,3,3	47.06	72.73	94.30	57.14	82.78
4,4	44.12	72.58	94.62	54.88	82.30
4,4,4	48.04	76.56	95.25	59.04	83.73
4,4,4,4	47.06	75.00	94.94	57.83	83.25
5,5	44.12	69.23	93.67	53.89	81.58
5,5,5	47.06	78.69	95.89	58.90	83.97
5,5,5,5	48.04	74.24	94.52	58.33	83.01

Table II reports results for the 128 filter configuration for the various multiple regions (same size) results. Similar results were obtained for 16, 32, and 64 filters multiple regions configurations. The best compromise was always between four and five. We also observed that the performance improved when increasing the number of filters and thus the number of features. However, of course, training a 16 filters on multiple regions (say two and two –overall 32 filters of size two and thus 32 features) was much faster that training the same configuration but with 128 per region. Last but not least, it was unclear, if increasing the number of features (i.e., number of filters per filter type) above say 500 would have given a improvement higher than 1% to 2% [4].

2) Training CNN: CNN code runs on top of TensorFlow[5]. TensorFlow is an open-source environment with a Python interface. To run our experiments we resorted on two systems, the details are summarized in the next paragraphs.

To select the most promising CNN architecture we used a multi-core system with 64 cores (Xeon E7-8867) each one capable to run a single thread; the multi-core machine has a memory of 500 GB. On this system we installed Python version 3.6 and TensorFlow 1.4. As anecdotal note, one leave-one-out experiment 16 filters, was requiring about twelve hours with an average of 10 to 20 parallel training processes. However, when going from 32 to 64 and 128 filters multiple windows we discovered that time and memory exploded. Digging into the issue we found two TensorFlow hidden parameters: *intra_op_parallelism_threads* and *inter_op_parallelism_threads*; they rule the level of parallelism. TensorFlow has two pools of threads, one for parallelizing inside one operation (i.e., the intra) and one to parallelize between operations (i.e., the inter). Suppose, we

[5]TensorFlow: www.tensorflow.org

have matrix multiplication, we can parallelize with multiple threads, (i.e., intra) but each thread get its one data copy. Since each thread has its own data copy, if the *intra_op_parallelism_threads* is not set, TensorFlow will attempt to use as much memory as possible and as many cores as possible. We were thus forced to set *intra_op_parallelism_threads* to ten and *inter_op_parallelism_threads* to zero to avoid processes to grow above limit (e.g., 100GB) and the use of all 64 processors, thus actually slowing down the experiments.

However, it was clear that training on the 1700 data points oracle would have required weeks and not days even on the 64 cores 500GB cluster. We thus resorted on the high performance servers of Calcul Quebec [6]. To speed up computation, we selected to run training on the nVidia K20 GPU capable of a peak of about 3.5 TFLops (Tera-Flops). As front-end we used just one node Intel Ivy Bridge EP E5-2650v2 and one core (RAM 4GB) per GPU card. On this hardware, the operating system is a CentOS 6.6, the Python version 3.5 and TensorFlow release is 1.0 with CUDA libraries 7.5. There are about 100 dual K20 GPU nodes. We decided to run 40 processes in parallel. This gave us the possibility to compute one leave-one-out experiment in about 12 hours.

F. Evaluation Method

We evaluate the performance of the models by using the following metrics:

1) Precision: Precision is defined as the percentage of methods or attributes predicted as being LAs that are correct with respect to the oracle.

2) True Positive Rate (TPR): TPR or relative recall is calculated as the ratio between the number of true positives and the total number of positive events. It indicates how many of the manually validated known LAs are correctly discovered.

3) F-Measure: F-Measure (F1 for short) is a measure used to combine the above two inversely related classification metrics, for example, precision and recall and is computed as their harmonic mean. It is defined as: $F = \frac{2 \cdot P \cdot R}{P+R}$

4) Area Under the Receiver Operating Characteristic (ROC) curve: ROC is a plot of the true positive rate against the false positive rate at various discrimination thresholds. The area under ROC is close to 1 when the classifier performs better and close to 0.5 when the classification model is poor and behaves like a random classifier.

5) Matthews Correlation Coefficient (MCC): MCC is a measure used in machine learning to assess the quality of a two-class classifier especially when the classes are unbalanced [28]. $MCC = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP+FP)(FN+TN)(FP+TN)(TP+FN)}}$

Values range from -1 to 1. Zero means that the approach performs like a random classifier. Other correlation values are interpreted as follows: $MCC < 0.2$: low, $0.2 \leq MCC < 0.4$: fair, $0.4 \leq MCC < 0.6$: moderate, $0.6 \leq MCC < 0.8$: strong, and $MCC \geq 0.8$: very strong [29].

[6]www.calculquebec.ca

G. Analysis Method

RQ1: ***Can we use CNN for linguistic smell detection?*** We construct an oracle of 1753 instances, 808 of which contain linguistic antipatterns, and use this as a dataset on which we train the CNN. We use leave-one-out validation as well as the various evaluation metrics described in section IV-F, to evaluate the performance. We previously determined the most promising configurations by splitting the training data into 90% for training and 10% for evaluating the parameters.

RQ2: ***Do traditional ML classifiers outperform CNN for linguistic smell detection?*** To answer this research question we first use part of speech, grammatical dependencies, semantic relations, and srcML tags as the set of features for our oracle. In Section II-B we list the six unique machine learning algorithms used in our approach. Two of the algorithms listed, SMO and SVM, are combined with different kernel implementations, which results in 9 unique machine learning algorithms that we tune as described in Sections II-B, IV-D and train on our oracle. We then use leave-one-out validation as well as the various evaluation metrics described in Section IV-F, to evaluate the performance of the classifiers. After evaluating the models, we compare their performance with the results found in **RQ1**.

RQ3: ***Are traditional ML classifiers faster than CNNs?*** To answer this research question we evaluate the top performing models in **RQ1** and **RQ2** with respect to the time taken to build one model.

V. Results

RQ1: Can we use CNN for linguistic smell detection?

Table III contains performance values of the CNN for various configurations, in terms of precision (PREC.), recall (REC.), F-measure (F1), ROC, and MCC computed using leave-one-out for various multi-filter configurations and filter numbers. Values for F1 range between 72% and 74%, 0.83 to 0.85 for ROC and 0.50 to 0.53 for MCC. The best preforming configuration, 128 filters of size 5 each, yields a recall of 73.51%, precision of 75.58%, and an F1 of 74.53%. ROC and MCC are 0.85 and 0.53 respectively, both of which indicate an adequate prediction model.

Clearly, CNN are able to provide satisfactory results in the context of lingustic smell detection. However, one may wonder if a complex and slow configuration such as three families of 128 filters of size 5 (i.e., 384 features) should be preferred over the faster 16 filters of type 2,3,4, and 5 (i.e., 64 features). The latter configuration differs from the former by only around 1% to 2% for all metrics.

RQ2: Do traditional ML classifiers outperform CNN for linguistic smell detection?

Table IV shows how both manually tuned and Bayesian optimization tuned traditional machine learning classifiers perform on the dataset. We vary the parameters of these ML classifiers as described in Section II-B. Table IV shows

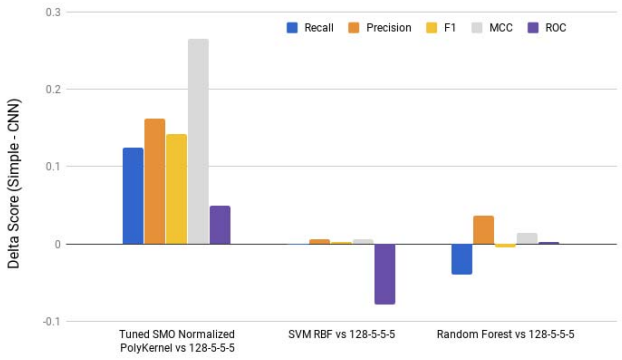

Fig. 2. Score Delta between traditional ML classifiers and CNN in Terms of Precision, Recall, F1-score, ROC, and MCC. Positive Values indicate better performance for traditional ML classifiers, negative values indicate better performance for CNN

TABLE III
CNN PERFORMANCE FOR VARIOUS CONFIGURATIONS

Configuration	REC.	PREC.	F1	ROC	MCC
16-2-3-4-5	72.51%	74.08%	73.29%	0.83	0.51
64-3-3-3	71.64%	73.85%	72.73%	0.83	0.50
16-3-3-3	71.39%	73.97%	72.66%	0.83	0.50
32-4-4-4	72.64%	75.16%	73.88%	0.83	0.52
128-5-5-5	73.51%	75.58%	74.53%	0.85	0.53

the results of these configurations with ROC values at least greater than 0.70. Considering the manually tuned classifiers, Random Forest and SVM RBF perform the best in terms of the evaluation metrics described. Random Forest is the best performing classifier in terms of precision (79.21%), ROC (0.85), and MCC (0.55). SVM RBF is the best performing classifier in terms of recall (73.42%) and F1 (74.76%).

By using Bayesian optimization via Auto-Weka, we achieve a traditional machine learning classifier that performs extremely well. Given the time, space, and model constraints, Auto-Weka determines the best model for the problem space, which is a tuned SMO Normalized Polykernel, giving a precision of 91.78%, F1 of 88.77%, ROC at 0.89, and MCC at 0.79. This classifier outperforms the manually tuned classifiers by more than 10% for precision, recall, F1, and MCC.

Figure 2 compares the results of the top performing manually tuned classifiers and the optimally tuned classifier relative to the top performing CNN configuration (128-5-5-5). A delta was calculated between the classifiers for each metric, per model. In this case, any bar above zero indicates that the traditional ML classifier performs better. Bars below zero indicate that CNN achieved a better performance. We find that traditional classifiers tuned manually achieve comparable results to CNN. However, if the effort is made to properly tune traditional classifiers with a method like Bayesian optimization, traditional machine learning techniques can dramatically outperform CNN. Although manual tuning of machine learning classifiers can improve performance in most cases, it requires expert knowledge, domain explicit thresholds, and sometimes even brute force approaches. As discussed by Fu and Menzies, this can be extremely costly and time consuming, and is therefore often not done properly, if at all, by researchers. The nature of the results obtained through Auto-Weka echoes those obtained through their DE optimization approach. Similarly, we do not suggest that Bayesian optimization is the best tuning approach for all software engineering tasks, rather we emphasize that traditional machine learning algorithms should be explored to their full potential before using more costly deep learning alternatives.

RQ3: Are traditional ML classifiers faster than CNNs?

To have an objective comparison of runtime for the two learning approaches, we provide the experimental environment as shown in Table VI. To obtain the runtime of the two approaches, we record the time taken to train just one model.

Overall results show that, even with relatively more powerful hardware, CNN requires quite substantial memory to train with reasonable times and is dramatically slower than traditional machine learning classifier alternatives, which achieve comparable, if not better, results in terms of precision and recall. Considering time taken to build just one model, traditional machine learning approaches are at least 16 times faster than the deep learning approaches. The discrepancy between tuning time is even larger; using Auto-Weka, the optimal model can be determined within as little as 2 minutes, compared to several days for the CNN with leave-one-out validation. Although one might argue that time can be seen as an up-front cost, the required memory by CNN remains an issue.
One may wonder if a 10-fold cross validation should have been preferred. Indeed, a single run would have been faster. However, it would have been necessary to repeat the experiment several times to establish performance boundary. Last but not least, results would not be as good for CNN as the training material would have been reduced.

VI. THREATS TO VALIDITY

Construct validity threats concern the relationship between theory and observation. In this study, construct validity threats are mainly due to measurement errors of labeled LAs. To build our oracle we used LAPD and randomly sampled methods from thirteen open-source Java systems. All LA instances were manually validated. The manual validation could be affected by subjectiveness or by the lack of domain knowledge. To mitigate those threats, in case of doubt, the surrounding code of an entity was analyzed and if the doubt still remained the entity was discarded. As for the recall, we are aware that the sample of instances that do not contain LAs may not be fully representative of the entire set. To mitigate this threat we randomly sampled source code entities from the entire pool of analyzed projects resulting in 945 entities that do not contain LAs from seven open-source projects. Note that the purpose of this training dataset is to have enough instances to train a model. Ideally, models are evaluated on a test set which is a

TABLE IV
TRADITIONAL ML CLASSIFIER PERFORMANCE

CLASSIFIER	PARAMETERS	RECALL	PRECISION	F1	ROC	MCC
Random Forest	#Trees 500	69.57%	79.21%	74.07%	0.85	0.55
	#Trees 1000	68.82%	78.69%	73.43%	0.85	0.54
J48 Pruned	Default	71.30%	73.40%	72.34%	0.76	0.49
	MinNumObj 8	67.20%	71.85%	69.45%	0.79	0.45
J48 UnPruned	Default	72.30%	73.30%	72.80%	0.78	0.49
SMO RBF	Cost 1, Gamma 0.1	66.46%	77.31%	71.48%	0.75	0.51
SMO PUK	Cost 5.0, Omega 8.0, Sigma 5.0	71.68%	77.24%	74.36%	0.77	0.54
SMO Poly Kernel	Cost 1.0, Exp 2.0	68.82%	68.14%	68.48%	0.71	0.41
	Cost 5.0, Exp 2.0	70.93%	68.80%	69.85%	0.72	0.43
SVM RBF	Gamma 0.1, cost 50	73.42%	76.16%	74.76%	0.77	0.54
SVM Poly	Cost 0.5, gamma 0.15	72.30%	75.10%	73.67%	0.76	0.52
Naive Bayes	Default	64.22%	65.11%	64.67%	0.72	0.35
Tuned SMO Normalized Poly kernel		85.96%	91.78%	88.77%	0.89	0.79

TABLE V
TIME AND MEMORY OF CNN AND TRADITIONAL ML CLASSIFIERS

Configuration	Time (s)	Memory (GB)
16-2-3-4-5	570.50	89.42
64-3-3-3	387.26	87.18
16-3-3-3	281.93	86.48
32-4-4-4	342.43	96.12
128-5-5-5	647.48	94.10
Random Forest	4.36	1.80
J48 Pruned	0.35	1.80
J48 UnPruned	0.09	1.80
SMO RBF	2.79	1.80
SMO PUK	3.62	1.80
SMO Poly Kernel	16.77	1.80
SVM RBF	0.48	1.80
SVM Poly Kernel	0.29	1.80
Naive Bayes	0.19	1.80
Tuned SMO Normalized Poly kernel	5.02	1.02

TABLE VI
COMPARISON OF EXPERIMENT ENVIRONMENT

Methods	OS	CPU	RAM
ML	Mac OS 10.12	Intel Core i7 4.01 GHz	64 GB
CNN	CentOS 7.3	64 Xeon E7-8867	500GB
	CentOS 6.6/ CUDA 7.5	40 x nVidia K20	40 x 4 GB

set of unseen data that represents the true prevalence of LAs in source code. However, for the purpose of our study, we only evaluate our models using leave-one-out validation. The reason being that our goal is to compare the performance of each approach not to claim the best accuracy.

Internal validity threats concern factors internal to our study that could have influenced our results. As explained in Section II-B, machine learning algorithms are trained with manual tuning of some parameters as well as Bayesian optimization through Auto-Weka. It is possible that better results could be obtained by providing Auto-weka more time and space resources. However, the worst case scenario would simply mean that our results represent a lower-bound. Different preprocessing and linguistic analysis techniques could be applied, and therefore it is possible that alternative representations and fact extractors could produce different results. However, our goal was to verify if simple, easy to gather information coupled with CNN could give better performance than a traditional machine learning approach. We cannot be sure that other types of feature will lead to different results; however such features will require heavier processing which defies the goal to benefit from a simple system architecture.

Conclusion validity threats concern the relationship between the treatments and the outcome. We report results using appropriate diagnostics for the machine learner performances (such as ROC and MCC). Then, when discussing findings we keep into account acceptable ranges for ROC and MCC (i.e., ROC must be ≥ 0.5 and MCC > 0). However, when comparing CNN with traditional machine learning we were not able to apply statistical tests. The reason is the time required to train CNN. In a sense this may hinder our conclusion on the accuracy but does not affect the huge difference in computation time. A further minor concern is the AUC comparison. TensorFlow CNN used in leave-one-out mode does not provide ROC and AUC. We thus mapped TensorFlow scores into probabilities by means of the softmax transformation and then used the score of one class to obtain ROC. On the computed ROC we did an approximated integration (i.e., sum of deltas FPR, TPR). This may result in an imprecision of AUC. We believe this is a minor point since we also computed precision, recall accuracy and MCC which are directly computed on the CNN classification.

Reliability validity threats concern the possibility of replicating this study. We attempt to provide all the necessary details needed to replicate our study and we share the oracle.

External validity threats concern the possibility of generalizing our results. Despite the fact we used thirteen open-source Java systems, and even though the thirteen systems cover different domains, we cannot guarantee that the finding generalize to the entire universe of Java programs. One further issue is the size of the dataset we used. Fu and Menzies [3] experiment was based on about 6,400 data points. Our results are reported on about 1,700 data points. The size of our dataset is tied to the need to manually validate LA instances. We are aware that 1,700 data points are half of the size of the smallest dataset used in previous studies such as [4], [5], [15]; due to this reason we opted for leave-one-out instead of a 10-fold cross validation. We plan to further extend the dataset to further validate our findings. However, this also will result in a longer computation time for CNN.

VII. Conclusion

In this paper, we perform a comparative study to investigate how traditional machine learning classifiers can outperform state of the art deep learning methods for predicting linguistic smells in source code. Results show that traditional machine learning algorithms run much faster, require far less tuning, can run on a standard laptop, and ultimately achieve similar or very comparable results to deep learning alternatives.

We find that, using Bayesian optimization techniques for hyper parameter tuning and model selection, we can achieve a traditional machine learning model that outperforms CNN in terms of all evaluation metrics used, as well as time and memory resources. Although deep learning is reported to produce results comparable and even superior to human experts for certain complex tasks, it doesn't seem to be a good fit for simple classification tasks like smell detection, and CNN in this case is significantly slower and more computationally expensive than the traditional machine learning classifiers. Therefore, researchers and practitioners should be careful when selecting machine learning models for their problems and be sure to explore the full potential of traditional machine learning models via hyper parameter tuning before turning to more complex and taxing approaches.

References

[1] J. Guo, J. Cheng, and J. Cleland-Huang, "Semantically enhanced software traceability using deep learning techniques," in *Proc. of the International Conference on Software Engineering (ICSE)*, 2017, pp. 3–14.

[2] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proc. of the International Conference on Automated Software Engineering (ASE)*, 2016, pp. 87–98.

[3] W. Fu and T. Menzies, "Easy over hard: A case study on deep learning," in *Proc. of the Joint Meeting on Foundations of Software Engineering (ESE)*, 2017, pp. 49–60.

[4] Y. Zhang and B. C. Wallace, "A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification," *CoRR*, 2015. [Online]. Available: http://arxiv.org/abs/1510.03820

[5] Y. Zhang and B. Wallace, "A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification," pp. 253–263, 2017.

[6] V. Arnaoudova, M. Di Penta, and G. Antoniol, "Linguistic antipatterns: What they are and how developers perceive them," *Empirical Software Engineering (EMSE)*, vol. 21, no. 1, pp. 104–158, 2015.

[7] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, 2016.

[8] D. Di Nucci, F. Palomba, R. Oliveto, and A. De Lucia, "Dynamic selection of classifiers in bug prediction: An adaptive method," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 1, no. 3, pp. 202–212, June 2017.

[9] S. L. Abebe, V. Arnaoudova, P. Tonella, G. Antoniol, and Y.-G. Guéhéneuc, "Can lexicon bad smells improve fault prediction?" in *Proc. of the Working Conference on Reverse Engineering (WCRE)*, 2012, pp. 235–244.

[10] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: An update," *SIGKDD Explorations Newsletter*, vol. 11, no. 1, pp. 10–18, November 2009.

[11] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, October 2001.

[12] E. J. Weyuker, T. J. Ostrand, and R. M. Bell, "Comparing the effectiveness of several modeling methods for fault prediction," *Empirical Software Engineering (EMSE)*, vol. 15, no. 3, pp. 277–295, June 2010.

[13] K. O. Elish and M. O. Elish, "Predicting defect-prone software modules using support vector machines," *Journal of Systems and Software (JSS)*, vol. 81, no. 5, pp. 649–660, 2008.

[14] B. Üstün, W. J. Melssen, and L. M. C. Buydens, "Facilitating the application of Support Vector Regression by using a universal Pearson VII function based kernel," *Chemometrics and Intelligent Laboratory Systems*, vol. 81, no. 1, pp. 29–40, 2006.

[15] Y. Kim, "Convolutional neural networks for sentence classification," *CoRR*, 2014. [Online]. Available: http://arxiv.org/abs/1408.5882

[16] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proc. of the International Conference on Neural Information Processing Systems*, ser. NIPS'12. Curran Associates Inc., 2012, pp. 1097–1105.

[17] K. A. Hallgren, "Computing inter-rater reliability for observational data: an overview and tutorial," *Tutorials in quantitative methods for psychology*, vol. 8, no. 1, p. 23, 2012.

[18] N. Madani, L. Guerrouj, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "Recognizing words from source code identifiers using speech recognition techniques," in *Proc. of the European Conference on Software Maintenance and Reengineering (CSMR)*, 2010, pp. 68–77.

[19] K. Toutanova and C. D. Manning, "Enriching the knowledge sources used in a maximum entropy part-of-speech tagger," in *Proc. of the Joint SIGDAT Conference on Empirical Methods in Natural Language Proc.essing and Very Large Corpora (EMNLP/VLC)*, 2000, pp. 63–70.

[20] M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini, "Building a large annotated corpus of English: the penn treebank," *Journal of Computational Linguistics*, vol. 19, no. 2, pp. 313–330, 1993.

[21] M. Collard, H. Kagdi, and J. Maletic, "An XML-based lightweight C++ fact extractor," in *Proc. of the International Workshop on Program Comprehension (IWPC)*, 2003, pp. 134–143.

[22] W. Fu, V. Nair, and T. Menzies, "Why is differential evolution better than grid search for tuning defect predictors?" *arXiv preprint arXiv:1609.02613*, 2016.

[23] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, no. Feb, pp. 281–305, 2012.

[24] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Advances in neural information processing systems*, 2012, pp. 2951–2959.

[25] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Auto-weka: Combined selection and hyperparameter optimization of classification algorithms," in *Proc. of the international conference on Knowledge discovery and data mining*, 2013, pp. 847–855.

[26] J. Bergstra, D. Yamins, and D. Cox, "Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures," in *International Conference on Machine Learning*, 2013, pp. 115–123.

[27] L. Kotthoff, C. Thornton, H. H. Hoos, F. Hutter, and K. Leyton-Brown, "Auto-weka 2.0: Automatic model selection and hyperparameter optimization in weka," *Journal of Machine Learning Research*, vol. 17, pp. 1–5, 2016.

[28] B. W. Matthews, "Comparison of the predicted and observed secondary structure of T4 phage lysozyme," *Biochimica et Biophysica Acta (BBA*, vol. 2, no. 405, pp. 442–451, 1975.

[29] J. Cohen, *Statistical power analysis for the behavioral sciences*, 2nd ed. Lawrence Earlbaum Associates, 1988.