Software Engineering: A Practitioner's Approach 9 th Edition

Preprint · January 2020

CITATIONS
0

READS
15,405

2 authors:

Terry Ruas
University of Göttingen
91 PUBLICATIONS **759** CITATIONS
SEE PROFILE

William Irvin Grosky
University of Michigan–Dearborn
248 PUBLICATIONS **3,176** CITATIONS
SEE PROFILE

Software Engineering: A Practitioner's Approach

9th Edition
By Roger Pressman and Bruce Maxim
ISBN10: 1259872971
ISBN13: 9781259872976
Copyright: 2020

Software Engineering: A Practitioner's Approach (SEPA), Ninth Edition, represents a major restructuring and update of previous editions, solidifying the book's position as the most comprehensive guide to this important subject. This text is also available in Connect. Connect enables the professor to assign readings, homework, quizzes, and tests easily and automatically grades and records the scores of the student's work.

LINK

PLEASE CITE IF YOU USE THIS MATERIAL

```
@incollection{Grosky:19,
  author       = {William Grosky and Terry Ruas},
  title        = {Data Science for Software Engineers},
  booktitle    = { Software Engineering: A Practitioner's Approach 9th Edition},
  publisher    = {MCGraw Hill},
  year         = 2019,
  editor       = {Roger Pressman and Bruce Maxim},
  pages        = {629-638},
  isbn         = {1259872971}
  address      = {The address of the publisher},
  edition      = 9,
  month        = 9,
  note         = {Appendix 2}
}
```

<EAP_NM>APPENDIX 2 (Pre-Print)

<eap_tt>Data Science for Software Engineers[1]

Pags. 629-638

<eapkt_tt>**Key Concepts**

Data science
Data science tools, libraries, and API's
Data science workflow
Machine learning
Statistical models
Regression problems
Classification problems
Dimensional reduction
Computational intelligence
Search-based software engineering

1. Data Science—The Big Picture

Data Science incorporates the work of many different disciplines to transform raw data into information, knowledge, and hopefully, into wisdom. Data science has a long history that incorporates concepts from computer science, mathematics, statistics, data visualization, along with algorithms and their implementations. It is beyond the scope of this Appendix to exhaustively detail all concepts under the rubric of "data science." Instead, we hope to provide a concise summary of the most important topics while connecting them to software engineering.

Figure 1 indicates that data science is the intersection of three major areas: computer science, mathematics and statistics, and domain knowledge (Conway, 2010).

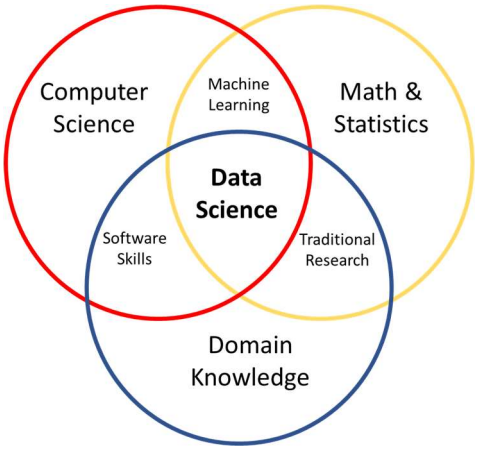


Figure 1. Data Science Venn Diagram.

[1] <bch_fn>This appendix has been contributed by William Grosky and Terry Ruas

A data scientist must be interested in more than the data itself. Using knowledge of mathematics and statistics along with domain specific knowledge, the data scientist develops necessary skills to evaluate whether data, experiments, and evaluation are properly designed for a given problem. However, to bring these capabilities to different scenarios requires a certain flexibility, and good computing skills can be the way to accomplish it.

1.1. Popular Languages, APIs and Tools

One of the great things about data science is that you can use it in virtually any environment that allows you to manipulate data. But to accomplish this, we need programming languages, APIs, and tools to make our lives easier. We'll provide an overview of these in the following sections.

1.1.1. Languages

When it comes to programming languages, we all have our biases towards the one we like the most. For practitioners in data science, this is no different. However, one should keep in mind that one size does not fit all, and the proper approach to language selection for data science applications is to choose the right tool for the right job, considering its constraints, contexts, and goals.

For data science application, fast prototyping is a strongly desired characteristic, one that allows us to produce interesting projects with a simplified and intuitive syntax. The resources available with respect to purpose and performance also play a crucial role in the adoption of a programming language. Thus, the less used a language is, the less attractive it will be for our daily tasks. Data science is closely related with data munging[2] — "the process of transforming and mapping data from one 'raw' data form into another format with the intent of making it more appropriate and valuable for a variety of downstream purposes, such as analytics." (Wikipedia). Data munging is a time-consuming activity and community support through well-documented APIs and libraries can make the entire difference, especially when looking for use examples, details about specific methods, constraints, and other technical aspects. Thus, a programming language used for data science applications should be broadly adopted, supported, and documented.

Therefore, it should come as no surprise that Python is widely used in the data science community. Other promising programming languages rising among data scientists are Scala and Julia, both more concerned with high performance and scalability. R is another interesting choice for data manipulation, specialized in statistical functions and data visualization libraries. Since its architecture is focused mainly on statistical analysis, data cleaning, and data visualization, R should not be your first choice for general purpose programming. In other words, R is highly effective, if used to solve the right problems.

Java popularity is indisputable in data science and many other areas of computer science. Following the recent trends in data science and big data, Java also has dedicated frameworks, such as Hive[3], Spark[4], and Hadoop[5]. Considering its non-specific architecture and verbosity, Java should not be the first option

[2] https://en.wikipedia.org/wiki/Data_wrangling
[3] https://hive.apache.org/
[4] https://spark.apache.org/
[5] https://hadoop.apache.org/

for advanced statistical analysis or data munging, especially for machine learning algorithms. For these cases, Python and R's dynamic scripting and huge dedicated libraries might be more interesting. Other strong programming languages, but not that popular among data scientists, are C/C++, F#, SQL.

1.1.2. Libraries and Tools

It is impossible to talk about data science without referring to the artifacts that assist us in the process of extracting knowledge out of data. In this section, we'll note some of the most popular ones offered in Python (VanderPlas, 2016).

NumPy is designed especially to efficiently manipulate n-dimensional arrays and handle scientific tasks. One can reshape the number of rows and columns, slice matrices, perform linear algebra operations, sort, search, and perform many other useful tasks. NumPy is used by a vast number of other libraries, and it is part of the SciPy stack.

There are two kinds of SciPy, the library itself and the scientific stack, composed of several open source ecosystems, including the former. The sub-libraries that form the scientific stack are: NumPy, SciPy, Matplotlib, IPython, Sympy and Pandas. The library, which is built on top of NumPy, is designed to provide efficient methods to deal with optimization, integration, and several other useful operations (Nunez-Iglesias, Walt, & Dashnow, 2017), (SciPy Developers, 2018).

As part of the scientific ecosystem, Pandas helps you with data structures and analysis through easy manipulations. It allows you to shape your data intuitively, providing easy adaptability from unstructured data to structured DataFrames. Some useful functions in Pandas include: indexing, labeling, fixing missing data records, and easy integration with different data structures (McKinney, 2017), (NumFOCUS, 2018).

Particularly important for data science, Python also has a large portfolio of machine learning libraries, in which Scikit-learn, TensorFlow, and Keras have a special place in the spotlight. Scikit-learn is probably one of the most known out-of-the-shelf machine learning libraries in Python, featuring several algorithm types, such as: clustering, regression, classification, and dimensionality reduction (Géron, 2017). TensorFlow, originally developed by the Google Brain team (part of Google's AI division), proposes an open source machine learning and deep learning framework for everyone.

Aside from the presented libraries, Python also has several other specialized tools that are used in data science, such as: (visualization) Matplotlib, Seaborn, Bokeh, and Plotly; (NLP) Natural Language Toolkit (NLTK), Gensim, spaCy, and Scrapy (ActiveWizards, 2018).

2. Data Science and Machine Learning

Data Science is an umbrella for a collection of data-driven approaches for finding approximate solutions to very difficult problems. The main enabling technology of data science is *machine learning*, a suite of statistics-based techniques that use an inductive approach which attempts to generalize from a set of known exemplars to unknown exemplars.

For example, our environment can consist of a set of multiple readings of various meteorological conditions, such as high-temperature, low-temperature, humidity, as well as several other values. We then have a small subset of these examples, our known exemplars, that are labeled; that is, for each example in this small set, the system is told whether it rained the following day or not. From this

training set of known exemplars, the system then constructs a mathematical model to categorize an unknown daily example as to whether or not it will rain the following day. Another example from software engineering would be to predict whether an individual piece of code has a fault, based on a training set of faulty and non-faulty programs. Alternatively, we might try to predict the cost of developing a new version of a program, based on the history of costs of previous versions of that same program.

In model building, there is an inherent tension between trying to construct a model which generalizes the training data, along with following the commonly held principle of Occam's razor, which is that the model should be as simple as possible to explain the current training set data. Figure 2 illustrates this conundrum. If the training set consists of only the circles, Occam's razor would choose the linear model, but with the addition of the triangle into the training set, perhaps Occam's Razor would choose the sin curve. So, which is the appropriate model: straight line, sin curve, or some, as yet, unknown curve?

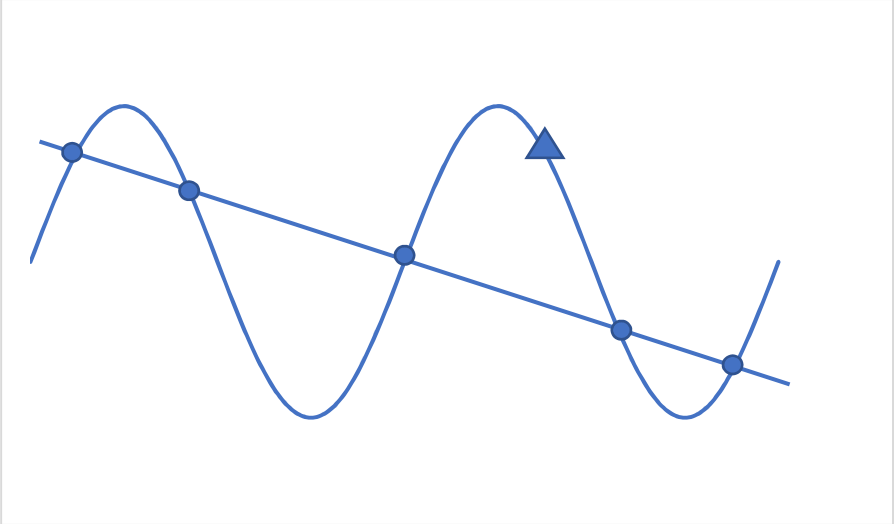


Figure 2 – Linear versus Non-Linear versus Non-Linear Model.

In the meteorological example, the system tries to discover commonalities among the meteorological readings, both in the training set for the days it rained the following day and for the days it didn't rain the following day, while also trying to discover differences among the meteorological readings in the training set between the days it rained the following day and the days it didn't rain the following day, using this metadata to determine whether it will rain or not tomorrow. Similarly, in the first software engineering example, the system tries to discover commonalities, in the training set, among programs with faults and those with no faults, as well as how faulty and non-faulty programs differ, eventually being able to determine whether an unknown program is faulty or not. In the second software engineering example, the system tries to find a general rule which connects the cost of succeeding versions of programs in the training set, using this rule to predict the cost of succeeding versions not in the training set.

The process which is followed during a data analytics project consists of (1) collecting the appropriate data, (2) *cleaning* the data, (3) *transforming* the data, (4) *analyzing* the data, and then (5) building an analytics-based *statistical model*.

1. **Data Collection.** Thinking about what data to collect is quite important, as it depends on the goal of your project. Questions that should be answered include what type of data is needed and how much data should be collected. For example, for software engineering data collection, what type of artifacts are needed? Do we need source code, object code, bug logs? What volume of data do I need to do the appropriate analytics?
2. **Data Cleaning.** Once the data is collected, it must be cleaned. This is a process which consists of eliminating problems in the data which would cause problems in further processing. For example, missing data should be filled-in, corrupt data should be found and corrected.
3. **Data Transformation.** After cleaning, it should be transformed to make it more suitable for the downstream analytics tasks. This process is called *data munging* or *data wrangling*. An example of this activity might be changing the format in which the data appears, eliminating punctuation in a text data file, and doing parts-of-speech analysis for text data.
4. **Data Analysis.** After all this, the data is ready to be analyzed and processed by various data analytics tools. But, before this happens, we generally use visualization tools for various tasks. For example, it may help us determine which features to use to predict the value of other features. It is only after this that we can determine the best analytical approach which can be used for predictive or inferential purposes.
5. **Data Set Fabrication.** Choosing an appropriate training set is important. The generalizations produced from different training sets might differ among themselves, but the hope is that the downstream answers produced are still correct. It is important not to *overfit* the training set, which means that the approach predicts items in the training set with close to 100% accuracy, but largely fails to predict the correct results for unknown items. This can happen quite easily if one is not careful to try to avoid this outcome. All this is determined by testing the derived statistical model on a *test set* of data to determine its error rate.

For the above process to work, objects must be represented by some mathematical structure which can be manipulated easily and compared among themselves. A common way of associating a mathematical structure with an individual object is to use *feature vectors*. A *feature* is a given property of an object. A feature vector is a vector of values for multiple features of an object class, so that the feature vectors for objects in the same object class have the same feature ordering. For example, the meteorological features of a day may have the following structure: (low-temperature, high-temperature, low-humidity, high-humidity, prevailing-cloud-type, overall-wind-strength). The variables of low-temperature, high-temperature, low-humidity, high humidity are *continuous* variables, while prevailing-cloud-type is an *unordered categorical variable* and overall-wind-strength is an *ordered categorical variable* (assuming the possible values are weak, average, strong).

In the software engineering environment, a feature vector corresponding to a piece of code could be a vector of values of several software metrics, such as number of lines of code, average program execution time, cohesion, coupling, and others. It is often challenging to choose the appropriate feature vector, and a new area of study, *feature engineering,* has evolved to help guide the process.

2.1 Machine Learning Approaches

Machine learning is an integral part of data science. The process discussed earlier in this section establishes the data set that is used to drive learning. *Supervised learning* consists of approaches where the user is in the loop and interacts with the learning system, mainly by providing certain types of meta-

information, such as labeled data for training sets. *Unsupervised learning* does not have the user in the loop to provide categorical information. These techniques are purely data driven and find ways of labeling the data from the data itself.

Within supervised learning approaches, there are two main types of problems: *classification* problems and *regression* problems.

Classification problems are those whose aim is to find to which of several classes an entity belongs; in other words, to predict a class label. A problem with two possible labels is called a *binary* classification problem, while a problem with more than two classes is called a *multi-class* classification problem. If an entity can fall into several classes, we have a *multi-label* classification problem. In this case, it often happens that the membership of an entity in an individual class is associated with a number between 0 and 1. This number can be interpreted as the strength of membership or the probability of membership. In this case, for a given entity, the sum of all its associated membership strengths or probabilities is equal to 1. A classic example of a binary classification problem is to classify email as *spam* or *non-spam*. An example of a multi-class classification problem would be to classify the contents of an email to various topic classes.

Regression problems are those whose aim is to predict the value of an output variable given the values of several input variables. The value predicted can be real-valued or discrete-valued. Suppose one had many feature vectors consisting of vita-related information for a prospective new hire. An example of a regression problem would be to predict the length of time that person will stay with your company before looking for a new job.

The boundary between classification problems and regression problems is imprecise. A regression problem where the values predicted are from a finite set can be couched as a classification problem where each class corresponds to a given value in the finite set of predicted values. Similarly, a classification problem can be couched as a regression problem where the output values predicted correspond to the set of class labels.

Popular techniques used for supervised learning include *linear regression*, *logistic regression*, *linear discriminant analysis*, *decision trees*, *k-nearest neighbor*, and *neural networks*. Popular techniques for unsupervised learning approaches include: *neural networks*, *clustering*, and *dimensional reduction*. We consider only a small sampling of these techniques in the Appendix.

2.1.1 Decision Trees

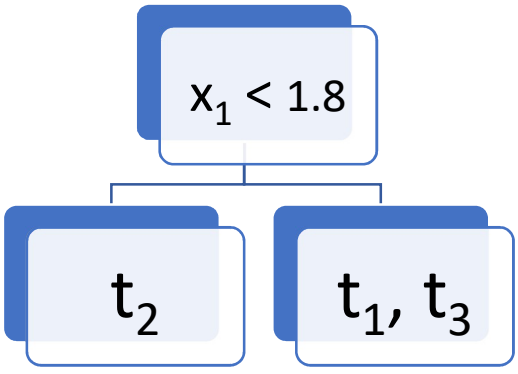
Decision tree learning is a predictive technique that uses data-derived observations contained in the branches of the tree to develop conclusions about a target value contained in the leaves of the tree. Based on the input variable values, a set of hierarchical decisions are made. The output variable value is found by following a tree from the root to a leaf, based on answers to questions asked along the way.

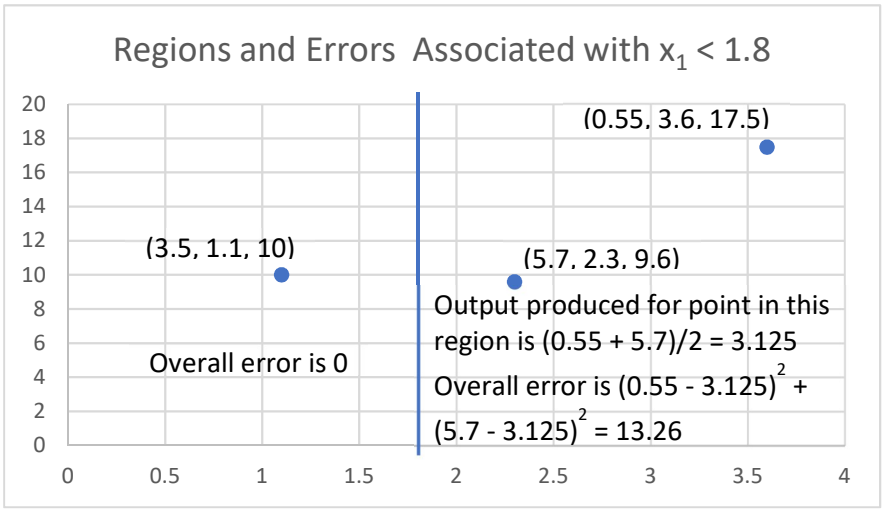
In general, decision trees can be binary or non-binary and the questions asked can be arbitrary, as long as they conform to the number of children at an individual node. For this Appendix, we will consider only binary trees with Boolean questions of the form $x < a$ or $x \leq b$, for some input variable x and constants a, b. If the answer to a question is TRUE, we would choose the left child to continue our walk down the tree, while if it is FALSE, we would choose the right child.

Given a training set of input and output variable values, we construct the tree choosing the type of Boolean question to ask at each internal node. This is usually done in a greedy fashion, simply asking, at a given node, which decision minimizes the sum of the squared errors. For visualization purposes, let us assume we have two input variables, x_1 and x_2, both of which are continuous. Suppose the training set is of the form (y,x_1,x_2) and is $t_1 = (5.7, 2.3, 9.6)$, $t_2 = (3.5, 1.1, 10)$, $t_3 = (0.55, 3.6, 17.5)$. We first must decide whether the first split will be on x_1 or x_2. We choose the input variable giving us the lowest error.

Now, each node of the tree is associated with a subset of the training set. For example, the root is associated with the entire training set. If the question at the root is $x_1 < 1$, then the root's left child is associated with the empty set and the root's right child is associated with the entire training set. However, if the question at the root is $x_1 < 1.8$, the root's left child is associated with t_2 and the root's right child is associated with the training tuples t_1 and t_3. Note that if we change 1.8 to 2.2, we would have the same association. However, even if the tree is the same, the choice of the split point would affect the results for input value pairs which are not in the training set.

So, what is the error produced by the $x_1 < 1.8$ split? If we stopped at this point, the tree would be used as follows. For an input value pair (c, d), if $c < 1.8$, we would predict the output value of 3.5, while if $c \geq 1.8$, we would predict the output value of 3.125, the average of 5.7 and 0.55. For this example, the squared error produced from the left child is 0, while the squared error produced from the right child is $(3.125-5.7)^2+(3.125-0.55)^2 \cong 13.26$. We could stop here, or perform a further refinement on the right-hand side, producing a division of the plane into 3 regions, each region associated with a single training set tuple. See Figure 3 for an illustration of the trees, associated regions, and errors for two different splits.





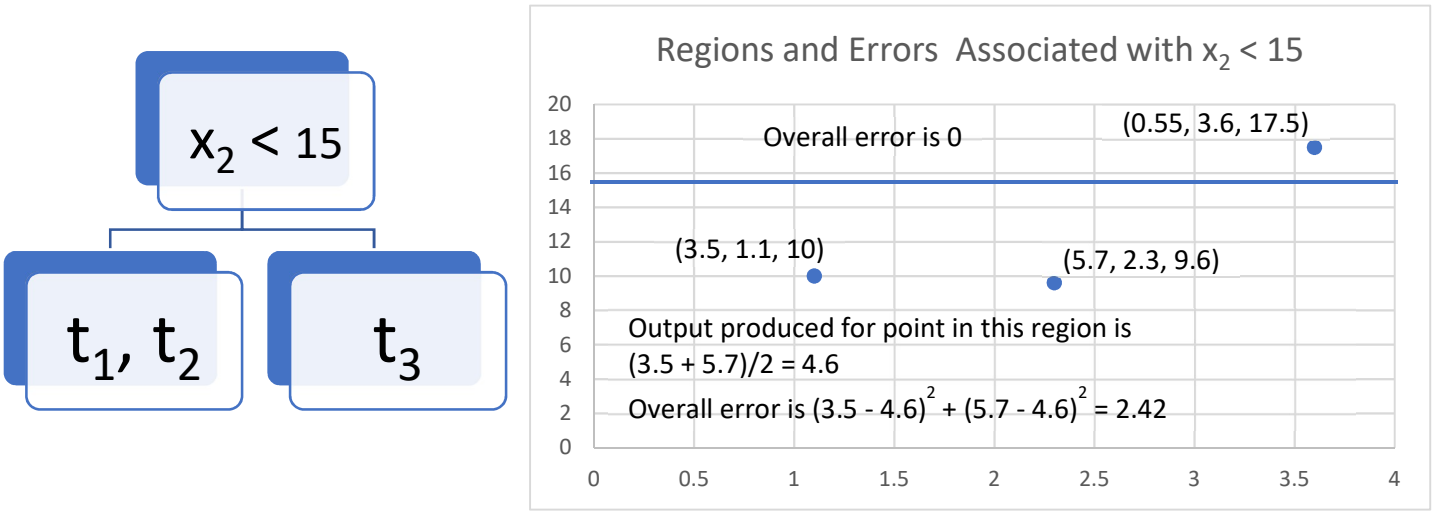


Figure 3 – Decision Trees Examples for Different Splits

There are many efficient approaches to finding the best tree, which include when and how a region should be split and when to cease splitting a region containing more than one training set tuple. Some of these approaches can be found in (James, Witten, Hastie, & Tibshirani, 2013).

In (Young, Abdou, & Bener, 2018), the use of decision trees in software engineering research is illustrated for the problem of just-in-time defect prediction. This technique predicts defects at small granularities. Code changes are predicted which are more likely to introduce defects. In this paper, it is demonstrated that the decision tree methodology is better than many other learning techniques for this problem. Decision trees are used in an *ensemble learning* environment. This type of learning combines many parallel learners in such a way that the final results are much improved.

2.1.2 Nearest Neighbor

The technique of k-nearest neighbor is an approach to estimate the probability of membership of an unclassified input variable tuple, *v*, in a finite set of classes. It is quite simple in its idea but can be very powerful. One finds the closest k points in the training set to the given point *v*. For a given class, *c*, suppose there are n points among these closest *k* training set points which belong to class *c*. Then the probability that *v* belongs to *c* would be *n*/*k*. If we had to label *v* with a single class, it would be the class with the highest probability. The value of *k* certainly affects the results. It has been found that values of k that are too small or too large don't perform well. As k increases to its sweet spot, the error decreases, but as k further increases, the error gets larger. See Figure 4 for an example: with 1-nearest neighbor, the black point is classified as red, with 3-nearest neighbors, it is classified as blue, and with 5 nearest neighbors, it is classified as red.

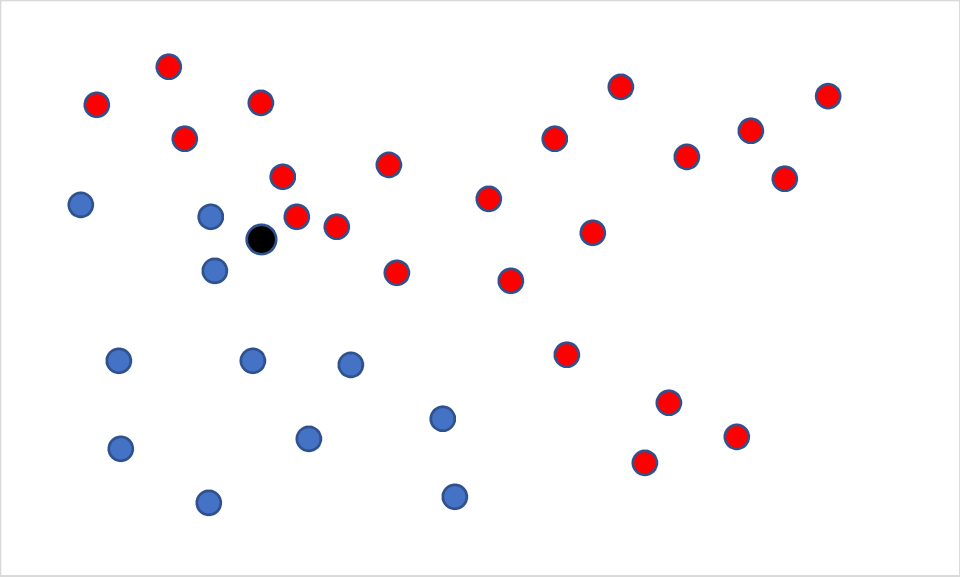


Figure 4 – k-Nearest Neighbor Example for k = 1,3,5.

In (Huang, et al., 2017), nearest neighbor is used to develop an improved approach for missing data in the software quality area. Missing data causes many problems for machine learning. There are many approaches to intelligently estimate values for this data.

2.1.3 Neural Networks

Neural networks embody *connectionism*, an architecture consisting of connections of multiple simple processors (i.e. brain cells) together in a massively parallel environment, supporting many concurrent processes, which can be used to solve many problems. The power of neural networks results from the fact that this approach models the output variable as a non-linear function of several linear combinations of the input variables. The more powerful neural networks have some form of feedback. To specify a neural network, we must specify the connectivity of the nodes, the way that a given node transforms all its inputs into an output, and how the final output is generated. In general, a neural network uses feedback through what is called a backpropagation technique (steepest descent or following the gradient direction) to train the network (estimate the parameters needed to carry out the regression). A weakness of neural networks is that it is hard to determine how particular parameters correlate with the parameters of the problem, leading to a weakness in explanatory power as to why a neural net has made a given decision.

2.1.4 Clustering

Clustering is a general data-driven approach to find groups of any entities that are similar in some sense. A group of *clusters* are found, each cluster containing a set of entities. The idea is that 2 entities in the same cluster are highly similar, while 2 entities, each in different clusters, are not as similar. It is generally up to the investigator to figure out what exactly is meant by similarity. In the context of this Appendix, the entities will be feature vectors and similarity is defined using a distance function between vectors. Vectors having a smaller distance between them will be more similar. There are hundreds of clustering algorithms, and there is no guarantee that you will get the same clustering from each of them.

Clusters can have different shapes, and some algorithms work better with convex shapes, while others can relax this condition. Some algorithms are specifically designed for high-dimensional spaces, while others aren't.

i. Dimensional Reduction

By dimensional reduction, we mean decreasing the lengths of the input feature vectors. In many important environments, the size of these vector can get quite large. In natural language processing, for example, each vocabulary word has its own position in the vector. It is quite common, therefore, for these vectors to have from 5,000 to 50,000 elements. Reducing the dimensionality would thus speed up the learning process. Initially, in several disciplines of computer science, this was the sole reason for dimensionality reduction. However, it was soon discovered that reducing the dimensionality of the input feature vectors also improved the performance of many of the underlying algorithms used in downstream applications.

3. Computational Intelligence and Search-Based Software Engineering

Computational Intelligence generally refers to the ability of a system (hardware and software) to learn a specific task from a set of data collected about that task. Some classify computational computing as a combination of granular computing (fuzzy sets, rough sets, probabilistic reasoning), neuro-computing (neural nets), evolutionary computing (genetic programming, genetic algorithms, and swarm intelligence), and artificial life (artificial immune systems). These approaches can be used for optimization, classification, search, and regression.

In search-based software engineering, computational intelligence-based techniques have been used for various sorts of optimizations. For example, there are papers (Ouni, Kessentini, & O Cinneide, 2017) using genetic algorithms for multi-objective optimization which quickly search the space of all possible code refactorings and recommend the best options, each option illustrating some particular trade-offs among the input variables, but still locally optimal. It has been shown that these approaches can also be used to build predictive models (Malhotra, Khanna, & Raje, 2017).

The merger of data science, machine learning, and search-based software engineering may lead to exciting breakthroughs in the way software is specified, designed, coded and tested. Time will tell.

Bibliography

ActiveWizards. (2018, February 13). *Top 20 Python libraries for data science in 2018*. Retrieved October 2018, from ActiveWizards: https://activewizards.com/blog/top-20-python-libraries-for-data-science-in-2018/

Conway, D. (2010, September 30). *The Data Science Venn Diagram*. Retrieved from Drew Conway Data Consulting: http://drewconway.com/zia/2013/3/26/the-data-science-venn-diagram

Géron, A. (2017). *Hands-On Machine Learning with Scikit-Learn and TensorFlow.* O'Reilly Media.

Huang, J., Keung, J., Sarro, F., Li, Y.-F., Yu, Y., Chan, W., & Sun, H. (2017). Cross-Validation Based K Nearest Neighbor Imputation for Software Quality Datasets: An Empirical Study. *The ournal of Systems and Software*, 226-252.

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). *An Introduction to Statistical Learning with Applications in R.* Springer.

Malhotra, R., Khanna, M., & Raje, R. (2017). On the Application of Search-Based Techniques for Software Engineering Predictive Modeling: A Systematic Review and Future Directions. *Swarm and Evolutionary Computation*, 85-109.

McKinney, W. (2017). *Python for Data Analysis Data Wrangling with Pandas, NumPy, and IPython.* O'Reilly Media.

NumFOCUS. (2018). *Python Data Anlysis Library*. Retrieved from pandas: http://pandas.pydata.org/

Nunez-Iglesias, J., Walt, S. v., & Dashnow, H. (2017). *Elegant SciPy.* O'Reilly Media.

Ouni, A., Kessentini, M., & O Cinneide, M. (2017). MORE: A Multi-Objective Refactoring Recommendation Approach to Introducing Design Patterns and Fixing Code Smells. *Journal of Software: Evolution and Process*, https://doi.org/10.1002/smr.1843.

SciPy Developers. (2018). *SciPy Library*. Retrieved from SciPy.org: https://scipy.org/scipylib/index.html

VanderPlas, J. (2016). *Python Data Science Handbook Essential Tools for Working with Data.* O'Reilly Media.

Young, S., Abdou, T., & Bener, A. (2018). A Replication Study: Just-In-Time Defect Prediction with Ensemble Learning. *Proceedings of the ACM/IEEE Sixth International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering* (pp. 42-47). Gothenburg, Sweden: ACM/IEEE.