ISSN 1870-4069

Runtime Prediction of Filter Unsupervised Feature Selection Methods

Teun van der Weij[1], Venustiano Soancatl-Aguilar[1], Saúl Solorio-Fernández[2]

[1] University of Groningen
Netherlands

[2] Instituto Nacional de Astrofísica, Óptica y Electrónica, Mexico

mailvanteun@gmail.com, v.soancatl.aguilar@rug.nl, ssolori1@asu.edu

Abstract. In recent years, the speed and quality of data analysis have been hindered by an increase in data size, an increase in data dimensionality, and the expensive task of data labeling. Much research has been conducted in the field of Unsupervised Feature Selection (UFS) to counteract this hindrance. Specifically, filter UFS methods are popular due to their simplicity and efficiency in counteracting performance problems in unlabeled data analysis. However, this popularity resulted in a great variety of filter UFS methods, each with their own advantages and disadvantages, making it hard to choose an appropriate method for a particular problem. Unfortunately, an inappropriate method choice can lead to a decrease in research or project quality, and it can render data analysis unfeasible due to time constraints. Importantly, terminating a method's analysis before completion means in most cases that no partial results are obtained either. Previous works on the evaluation of filter UFS methods focused mainly on assessing clustering and classification performance. Although very useful, choosing an appropriate method often requires knowledge about the method's runtime as well. In this paper, we study the runtimes of six popular filter UFS methods using synthetic and real-world datasets. Runtime prediction models were trained on 114 synthetic datasets and tested on 29 real-world datasets. The models showed good performance on four out of the six methods. Finally, we present general runtime guidelines for each method. To the best of our knowledge, this is the first paper that investigates methods' runtimes in this fashion.

Keywords: Feature selection, unsupervised feature selection, runtime prediction, execution time prediction, filter methods.

1 Introduction

Feature Selection, also known as Attribute or Variable Selection, concerns selecting a subset of the most relevant features from a dataset. Selecting the

most relevant features can be useful to achieve three main goals: improve prediction accuracy, faster predictions, and a better understanding of the phenomena that the data represent [1]. The importance of Feature Selection increases as the data grows in the number of objects and especially in the number of features, yielding all sorts of problems relating to the "curse of dimensionality" [2]. A high number of features requires more computational resources; if many features need to be analyzed, the speed of both the training and the predictions of a learning algorithm decrease. Furthermore, an excess of features reduces generalization capabilities and may negatively affect predicting performance [3]. Additionally, it is harder to understand the underlying mechanisms that the data describes when many irrelevant features clutter the relevant ones [1]. Feature Extraction is another closely related dimensionality reduction strategy with similar advantages to Feature Selection. However, Feature Extraction, which includes methods such as principal component analysis, unclearly transforms the relevant features, complicating the interpretation of the data [4, 5].

According to the availability of information in the data, datasets can be classified as completely labeled, partially labeled, or completely unlabeled. Fully labeled datasets require supervised methods, partially labeled require semi-supervised methods, and unlabeled datasets require unsupervised feature selection methods. The labels of objects in a dataset can be categorical, ordinal, or continuous [6]. These labels can, for example, describe what kind of animal the features represent, the place a bowler got in a bowling competition, or how happy a person says she is. Such labels are often not available, especially where high-dimensional data is present, such as in text mining, bioinformatics, and social media [3, 7]. Moreover, data labeling is expensive in both time and money because the labels need to be accurate, requiring qualified human labor [8]. Therefore, for unlabeled data, Unsupervised Feature Selection (UFS) methods are often used. Other important advantages of UFS methods include that these methods perform well when prior knowledge is unavailable and that they are less prone to overfitting [1]. UFS methods can be subdivided further into three categories: filter, wrapper, and hybrid methods [6, 9]. Filter methods are the fastest and most scalable methods, and they work independently of the classifier. Wrapper methods use a classifier or learning algorithm to evaluate a subset of features, which generally makes it much more computationally expensive. Moreover, wrapper methods need to be entirely retrained when a different classifier is used. Hybrid (embedded) methods aim to be a mixture of filter and wrapper methods, trying to balance the two approaches to get the benefits of both [9]. However, the integration of filter and wrapper approaches is generally insufficient, leading to lower classification performance [7].

Because of the advantages of the filter approach, many methods have been developed in this UFS category [6]. As a consequence, choosing an appropriate method for the task at hand can be time-consuming and difficult. A choice of a UFS method is important because of two reasons. First, one wants to obtain

the best possible insight from the data, which entails optimal understanding, optimal clustering and classification performance. Missed or uncertain insights might result in less fruitful research. Second, there is limited time available for all research. Depending on how limited the time is, a method must be selected that operates within these time constraints. Problems arise especially if it is unknown a priori how long these methods take to analyze a dataset. The runtime of a method analyzing a certain dataset could take several days, and larger datasets might take months or longer, even with fast hardware and software. The setup of a research project must be adjusted to the runtime of a method, which can mean sacrificing clustering and classification performance for runtime gains.

Solorio-Fernández *et al.* [7] saw the lack of and need for a comprehensive empirical study to enable users to choose an appropriate filter UFS method. The authors systematically analyzed the performance of 18 filter UFS methods, which were applied on 75 datasets. They also scored the methods based on clustering and classification performance. Consistent with the literature, the authors found that statistical-based methods generally had the worst clustering and classification performance, but they were the quickest methods. On the other hand, multivariate spectral/sparse-learning-based methods had significantly higher scores for clustering and classification, but they were substantially time-consuming. Furthermore, Solorio-Fernández *et al.* [7] reported the runtimes for every method ran on a dataset, which illustrated that some methods analyze a dataset in fractions of a second and some take more than seven days. As time constraints affect research quality, and Solorio-Fernández *et al.* [7] showed that there is a high variation in runtimes between methods, further research on method runtimes is needed to make a good a priori decision for a certain UFS method. Additionally, the number of objects also affects the runtime, as methods need to analyze more data. Moreover, datasets with millions and trillions of features already exist, for example, the MovieLens dataset (over 20 million objects) and the Google Books Ngram dataset (over 10 billion objects) [10, 11]. Furthermore, the feature sizes are very likely to further increase according to Bolón-Canedo *et al.* [12]. So, even if the runtimes shown by Solorio-Fernández *et al.* [7] are not problematic with maximums of 12960 objects and 2283 features, runtime problems are bound to arise with much bigger dataset sizes. Moreover, terminating a running method before completion means that no partial results can be obtained unless complicated changes to the methods are made.

To help users choose an appropriate method with respect to these runtimes issues, we investigate six popular UFS methods by predicting their runtimes based on the number of objects and features of a dataset. As a result, we contribute to the runtime knowledge of filter UFS methods by providing prediction models and general runtime guidelines. We examine the runtime performance of the six filter UFS methods available in the scikit-feature package created by Li *et al.* [3], which contains the implementation of some classical, relevant and more cited methods in the literature. We now present a brief overview of these six methods. The runtime prediction of the six methods will be discussed in the Methodology section.

The rest of the paper is organized as follows: Section 2 describes the filter UFS methods analyzed in this study. Section 3 describes the evaluation methodology used in our experiments. Section 4 reports the experimental results. Section 5 discusses the main insights and the general runtime observations derived from our experiments. Finally, Section 6 concludes the paper and provides some directions for future work.

2 Filter UFS Methods

According to Alelyani *et al.* [4] and Solorio-Fernández *et al.* [6, 7], filter UFS methods can be categorized into univariate and multivariate methods. We describe the key characteristics and the corresponding methods of both categories in Sections 2.1 and 2.2.

2.1 Univariate Methods

Univariate methods evaluate features separately and score a feature based on a certain criterion. Consequently, these methods do not have to solve the computationally expensive combinatorial optimization problem of selecting a feature subset [13]. Therefore, relevant features are found relatively quick. However, redundant features (those highly similar to other features) cannot be filtered out because features are not compared to other features, potentially leading to superfluous features in the selected set of features.

Low Variance: This relatively simple method ranks the features based on their variance [5, 14]. The underlying idea is that features that differ more in value are more relevant to uncover the underlying mechanisms in a dataset and to help differentiate instances between different classes [3]. Features with a low variance often do not carry much relevant information and do not differentiate between classes [7].

Laplacian Score: This method developed by He *et al.* [15] scores the importance of a feature by analyzing how well it preserves the locality. The Laplacian matrix is derived from the distance between data points, so the method can capture and analyze the local structure in the data space, which is often more important than the global structure [15]. Each feature is individually scored, and the top k features with the lowest Laplacian Score are selected [3].

SPEC: SPECtrum decomposition, created by Liu *et al.* [5], extends on the Laplacian Score method and is also built on a similar idea: "a feature that is consistent with the data manifold structure should assign similar values to instances that are near each other" [3]. This method ranks the features based on a consistency score calculated by three different criteria [5].

2.2 Multivariate Methods

A multivariate approach entails that subsets of the feature set are evaluated and scored together. Because of this, they are able to filter out both irrelevant and redundant features. However, selecting a (sub)optimal subset of a set of features is computationally expensive, as illustrated by the "subset sum" problem [16]. Even though the multivariate methods try to approach this problem efficiently, multivariate methods are generally much slower than univariate methods [4, 6, 7].

MCFS: The Multi-Cluster Feature Selection method developed by Cai *et al.* [13] selects features "that can cover the multi-cluster structure of the data where spectral analysis is used to measure the correlation between different features" [3]. As with previous methods, a Laplacian Matrix is constructed. The MCFS method takes the first k eigenvectors of this Laplacian matrix and calculates the importance of features by a regression model with l_1 norm regularization [13]. After solving the regression problems, a coefficient is computed where a high MCFS score means that the feature is important [3].

UDFS: Yang *et al.* [17] propose the Unsupervised Discriminative Feature Selection method, which uses both the discriminative information and feature correlations to select features [3, 7]. UDFS efficiently optimizes the $l_{2,1}$ norm regularized minimization problem with orthogonal constraint Yang *et al.* [17]. The UDFS method trains a linear classifier which obtains the highest local discriminative score for all features [3]. As with MCFS, the higher the score, the more important the feature is [17].

NDFS: The Nonnegative Discriminative Feature Selection method by Li *et al.* [18] first uses spectral analysis with nonnegative and orthogonal constraints to learn pseudo-class labels, and these labels are defined as nonnegative real values. Afterwards, the authors introduce a novel iterative algorithm to efficiently solve the $l_{2,1}$ norm regularization problem NDFS creates. The top k features that most relate to the pseudo-class labels are selected [18].

3 Methodology

The general protocol of our study is as follows: First, we measured the runtime of the six UFS methods on 114 synthetic datasets. Then, we used four different regression models to fit these runtimes based on the number of objects and features of a dataset. Subsequently, we evaluated the performance of these regression models using 10-fold cross-validation. Finally, we tested the best method per model on 29 real-world datasets.

First, we discuss the synthetic and the real-world datasets. Second, we elaborate on the four regression models and how we evaluate them. Lastly, we discuss the software and hardware specifications used in this study, including the parameter settings for the methods, the way of measuring runtimes, and details on the CPUs.

3.1 Datasets

To construct good models, good training data is needed. In our case, good training data means a considerable number of datasets with spread-out dimensions. Runtime patterns become more apparent for both humans and runtime prediction models when they evenly cover a larger part of the space of the number of objects and features. Therefore, to get training data that meets these conditions, we generated synthetic datasets SD with stepwise differing dimensions in both the number of objects and the number of features. These dimensions range from 500 objects and 500 features until, but not including, 10,000 objects and 10,000 features with a step size of 500. However, we constrained the maximum size per dataset to 10,000,000 data points. The number of data points is estimated as the product of the number of objects and the number of features. This means that datasets were only generated when this product is lower than or equal to the maximum size. Mathematically:

$$\begin{aligned} SD = \{D_{ij} \mid & i = 500, 1000, 1500, \ldots, 9500; \\ & j = 500, 1000, 1500, \ldots, 9500; \\ & i \times j \leq 10,000,000\}, \end{aligned} \tag{1}$$

where D_{ij} represents a dataset composed of i objects and j features. For example, a dataset of 8,000 objects and 4,000 features would not be generated, but a dataset with 4,000 objects and 2,500 features would have been generated. This resulted in 114 synthetic datasets in total.

The maximum size constraints for objects and features were chosen to keep the runtimes within the time and resource limits of the study. Furthermore, the datasets were generated with certain parameters, which are described in detail in Table 1. Note that the hypercube size value indicates the multiplication factor of the hypercube. This factor influences the spread of clusters/classes, which might make it easier for methods to converge. The effect of the hypercube size multiplication factor on the runtime is not part of our research. However, we varied this factor to improve the generalization of our runtime prediction models. Furthermore, the dataset generation parameter values were designed in a way to mimic real-world datasets. As a last note, all the default settings[3] were used for the parameters not described in the Table 1.

In addition to the synthetic datasets, we tested the runtime prediction models on real-world datasets to verify their performance. These were taken from the ASU Feature Selection Repository [3]. These datasets are all different types of data: text, face images, handwritten images, biological, amongst others. Further details of these real-world datasets can be found in Table 2. Both the synthetic and the real-world datasets were standardized to have a mean of 0 and a standard deviation of 1, as recommended by [14].

[3] Further general description and default settings of the parameters can be found on https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_classification.html.

Table 1. Synthetic dataset details.

Index	Objects	Features	Informative features	Redundant features	Classes	Clusters per class	Randomly labeled	Hypercube size
1	500	500	231	43	28	3	0.018	1.533
2	1000	500	221	174	19	1	0.435	0.615
3	1500	500	89	71	27	1	0.222	2.451
4	2000	500	8	171	3	3	0.256	2.365
5	2500	500	141	116	10	2	0.167	2.008
6	3000	500	189	144	27	1	0.225	0.872
7	3500	500	8	31	29	1	0.154	1.247
8	4000	500	112	163	3	2	0.258	0.987
9	4500	500	110	5	11	1	0.262	1.234
10	5000	500	83	89	18	3	0.416	0.917
11	5500	500	165	176	10	2	0.338	1.693
12	6000	500	158	221	18	3	0.137	2.032
13	6500	500	183	227	27	1	0.281	1.431
14	7000	500	75	23	3	2	0.189	1.319
15	7500	500	121	181	10	3	0.465	1.470
16	8000	500	179	196	3	2	0.049	1.185
17	8500	500	183	78	13	1	0.470	2.369
18	9000	500	122	78	6	3	0.497	1.009
19	9500	500	107	238	3	2	0.286	1.633
20	500	1000	58	291	3	3	0.278	1.384
21	1000	1000	333	140	8	3	0.371	1.213
22	1500	1000	98	361	16	2	0.467	2.117
23	2000	1000	85	128	23	2	0.437	0.744
24	2500	1000	357	406	8	1	0.375	1.575
25	3000	1000	262	46	16	1	0.323	0.886
26	3500	1000	481	188	13	1	0.020	0.946
27	4000	1000	478	344	8	1	0.088	0.747
28	4500	1000	38	31	2	2	0.310	0.808
29	5000	1000	373	466	22	3	0.065	1.299
30	5500	1000	125	334	30	1	0.040	0.736
31	6000	1000	319	196	22	3	0.232	0.675
32	6500	1000	387	472	12	1	0.348	0.608
33	7000	1000	429	294	10	3	0.097	2.313
34	7500	1000	69	200	18	1	0.059	0.737
35	8000	1000	195	181	16	2	0.045	0.546
36	8500	1000	184	353	8	3	0.013	0.837
37	9000	1000	370	113	10	2	0.420	2.442
38	9500	1000	322	485	21	3	0.059	1.728
39	500	1500	541	182	24	1	0.049	1.962
40	1000	1500	398	689	18	2	0.072	0.687
41	1500	1500	288	138	2	1	0.117	1.993
42	2000	1500	190	564	8	2	0.459	2.122
43	2500	1500	184	330	20	3	0.485	0.568
44	3000	1500	703	285	12	3	0.243	2.411
45	3500	1500	312	598	17	2	0.305	2.156

Table 1 continued from previous page.

Index	Objects	Features	Informative features	Redundant features	Number of classes	Clusters per class	Randomly labeled	Hypercube size
46	4000	1500	654	118	11	2	0.286	1.310
47	4500	1500	412	31	24	1	0.267	2.109
48	5000	1500	581	455	9	3	0.288	1.150
49	5500	1500	219	73	30	3	0.223	1.824
50	6000	1500	51	504	6	3	0.083	1.339
51	6500	1500	656	586	4	3	0.313	1.653
52	500	2000	546	537	9	1	0.108	0.502
53	1000	2000	379	388	25	1	0.303	1.181
54	1500	2000	217	776	10	1	0.075	0.620
55	2000	2000	726	721	25	2	0.439	1.016
56	2500	2000	829	711	3	3	0.451	0.669
57	3000	2000	485	258	22	3	0.477	1.955
58	3500	2000	778	506	29	1	0.248	2.167
59	4000	2000	413	495	19	3	0.467	1.704
60	4500	2000	238	526	19	1	0.200	1.840
61	5000	2000	238	514	29	2	0.234	1.230
62	500	2500	606	677	20	3	0.227	1.454
63	1000	2500	1056	1025	21	3	0.037	2.085
64	1500	2500	1116	990	8	3	0.433	1.157
65	2000	2500	623	1208	16	2	0.296	0.744
66	2500	2500	501	757	8	1	0.379	1.178
67	3000	2500	257	468	5	3	0.221	0.800
68	3500	2500	1191	840	10	1	0.499	2.377
69	4000	2500	399	281	9	1	0.470	2.127
70	500	3000	826	732	19	3	0.139	2.468
71	1000	3000	654	245	16	3	0.422	1.896
72	1500	3000	646	967	6	1	0.339	2.058
73	2000	3000	765	631	25	2	0.011	2.286
74	2500	3000	1048	668	19	1	0.236	1.702
75	3000	3000	1006	1140	16	3	0.261	2.370
76	500	3500	1445	41	10	3	0.309	2.364
77	1000	3500	26	1685	17	1	0.134	1.301
78	1500	3500	574	451	17	1	0.269	1.729
79	2000	3500	1047	280	8	1	0.051	1.645
80	2500	3500	162	407	21	2	0.192	1.441
81	500	4000	1445	1043	29	2	0.243	2.244
82	1000	4000	233	57	27	2	0.063	0.848
83	1500	4000	1474	222	11	3	0.453	1.054
84	2000	4000	1727	488	20	1	0.067	2.186
85	2500	4000	1387	1555	8	3	0.389	1.156
86	500	4500	73	1693	6	2	0.198	2.403
87	1000	4500	75	1074	3	2	0.364	1.640
88	1500	4500	699	1067	7	1	0.125	0.544
89	2000	4500	839	209	21	3	0.257	1.706
90	500	5000	1496	1588	2	2	0.058	0.789
91	1000	5000	2122	1717	4	3	0.499	0.654

Table 1 continued from previous page.

Index	Objects	Features	Informative features	Redundant features	Number of classes	Clusters per class	Randomly labeled	Hypercube size
92	1500	5000	2357	1980	4	2	0.164	0.972
93	2000	5000	384	1143	14	2	0.295	2.265
94	500	5500	875	1162	3	3	0.028	1.483
95	1000	5500	2526	77	3	1	0.217	1.124
96	1500	5500	1439	1493	23	2	0.475	1.664
97	500	6000	2028	879	28	2	0.497	1.459
98	1000	6000	933	2944	5	3	0.068	1.112
99	1500	6000	2248	2935	3	2	0.179	1.283
100	500	6500	480	636	27	2	0.255	1.326
101	1000	6500	453	3142	29	2	0.206	1.425
102	1500	6500	2920	2502	20	1	0.402	1.608
103	500	7000	1317	1030	25	3	0.068	1.294
104	1000	7000	45	589	22	2	0.060	1.066
105	500	7500	187	3440	18	3	0.037	1.815
106	1000	7500	2835	323	18	2	0.334	1.335
107	500	8000	266	690	21	2	0.470	1.573
108	1000	8000	2605	2334	14	1	0.434	0.574
109	500	8500	2630	2774	25	3	0.440	1.432
110	1000	8500	3451	3706	23	1	0.037	0.742
111	500	9000	661	2287	18	3	0.337	1.524
112	1000	9000	2150	3700	19	3	0.266	2.206
113	500	9500	4749	3582	14	3	0.482	1.719
114	1000	9500	4330	2531	19	2	0.127	1.931

3.2 Runtime Prediction Models

For our experiments, we use four models, namely simple linear regression, multiple linear regression, power regression, and exponential regression. We use these relatively simple models for the three following reasons:

1. The number of objects and the number of features of a dataset are the only two independent variables in our study. Therefore, models that sophistically select or independently weight variables are excessive.
2. We assume that users generally want to know a good runtime approximation of a method in terms of seconds, hours, days, months, or years. Therefore, a runtime approximation would be sufficient to help users in choosing an appropriate filter UFS method, which simpler models can give. How many seconds or days exactly it will take will likely be less relevant.
3. Precise runtime prediction of UFS methods running in different environments is out of the scope of this paper because it is expensive in both time and hardware resources. Runtime predictions vary based on the environment in which the methods are run due to different hardware arrangements and other tasks being run in that environment. Simple models can more easily use and adapt to their own environment.

Table 2. Real-world datasets.

Index	Name	Number of objects	Number of features	Number of classes
1	Isolet	1560	617	26
2	Yale	165	1024	15
3	OLR	400	1024	40
4	WarpAR10P	130	2400	10
5	Colon	62	2000	2
6	WarpPIE10P	201	2420	10
7	Lung	203	3312	5
8	COIL20	1440	1024	20
9	Lymphoma	96	4026	9
10	GLIOMA	50	4434	4
11	ALLAML	72	7129	2
12	Prostate-GE	102	5966	2
13	TOX-171	171	5748	4
14	Leukemia	72	7070	2
15	Nci9	60	9712	9
16	Carcinom	174	9182	11
17	Arcene	200	10000	2
18	Orlraws10P	100	10304	10
19	Pixraw10P	100	10000	10
20	RELATHE	1427	4322	2
21	PCMAC	1943	3289	2
22	BASEHOCK	1993	4862	2
23	CLL-SUB-111	111	11340	3
24	GLI-85	85	22283	2
25	SMK-CAN-187	187	19993	2
26	USPS	9298	256	10
27	Madelon	2600	500	2
28	Lung-small	73	325	7
29	Gisette	7000	5000	2

The three reasons above-mentioned lead us to use simple models that are easy to understand. From the simpler models, we selected those which were expected to perform well based on visual inspection of the runtimes. Additionally, we selected models based on the time complexity of a method, which were only available for the SPEC and MCFS methods.

In the following subsections, we describe the runtime prediction models used in our experiments. It is important to mention that for simple and multiple linear regression models, the objective functions were given, whereas we merely present the model's predicted runtime per sample for power regression and exponential linear regression. Moreover, for all models, we do not estimate a y-intercept because we expect the runtime to approach 0 when the number of objects and features approach 0. Using a y-intercept might result in overfitting on the training data and worse values for the remaining parameters.

Lastly, all the runtimes and the corresponding errors are presented and calculated in seconds throughout the whole paper.

Simple linear regression Let y_i and βx_i ($i = 1, ..., n$, with n denoting the number of samples) be the true and the fitted runtime, respectively. A sample consists of a dataset and the corresponding true runtime of one method. The criteria of fitting S is when the sum of squared residuals of the linear regression model is minimal, i.e.:

$$S(\beta) = \sum_{i=1}^{n} \hat{\varepsilon}_i^2 = \sum_{i=1}^{n} (y_i - \beta x_i)^2, \tag{2}$$

where the coefficient β is the slope of the linear regression line tuned to minimize the residual sum of squares between the true and fitted runtimes, $\hat{\varepsilon}_i^2$ denotes the squared fit error of sample i, and x_i is defined as the product of the number of objects and features of the dataset of sample i (the total number of data points of a dataset).

Multiple linear regression Similar to Equation 2, the objective function S of the multiple linear regression model is defined as:

$$S(\beta_1, \beta_2) = \sum_{i=1}^{n} \hat{\varepsilon}_i^2 = \sum_{i=1}^{n} (y_i - \beta_1 x_{i1} - \beta_2 x_{i2})^2, \tag{3}$$

where the coefficients β_1 and β_2 denote the corresponding slopes of the regression lines of x_{i1} and x_{i2} which are fitted to minimize the residual sum of squares, y_i and $\hat{\varepsilon}_i^2$ are defined the same as in Equation 2, and x_{i1} denotes the number of objects and x_{i2} represents the number of features of the sample dataset i.

Power regression The power regression model best models situations where the runtime equals the independent predictor variables raised to a power. As previously described, Equation 4 represents the fitted runtime of one sample. Consequently, the power regression model is defined as:

$$\hat{y} = \beta_1 x_1^{\beta_2} x_2^{\beta_3}, \tag{4}$$

where $\hat{y}$ denotes the fitted runtime, x_1 and x_2 denote the number of objects and features of a dataset, respectively, and the parameters β_1, β_2 and β_3 are minimized with the Trust Region Reflective algorithm [19].

This function is inspired by both visual inspection of the method runtimes and the time complexity of the SPEC and MCFS methods. It allows different effects of both object and feature numbers through β_2 and β_3, but they still influence each other because they are multiplied. In other words, the effect of the number of objects on the runtime is partially determined by the number of features and vice versa.

Exponential regression This model combines exponential and linear regression. As with Equation 4, Equation 5 represents the fitted runtime. The Exponential and Linear regression model is defined as:

$$\hat{y} = \beta_1 e^{(\beta_2 x_1)} \beta_3 x_2, \tag{5}$$

where $\hat{y}$ denotes the fitted runtime, x_1 can either be the number of objects or the number of features of a dataset and x_2 is the remaining option, and β_1, β_2 and β_3 are minimized with the Trust Region Reflective algorithm.

The design of this model is set up to let x_1 have a strong exponential influence on the runtime prediction and x_2 to have a secondary linear role. These roles are inspired by the plots presented in the results section and more detailed investigation of the effect on the runtime by only changing either objects or features sizes. From now on, we refer to the exponential regression model with x_1 denoting the number of objects and x_2 denoting the number of features as $\text{Exp}_{\text{objects}}$. Similarly, we refer to the exponential regression model with x_1 denoting the number of features and x_2 denoting the number of objects as $\text{Exp}_{\text{features}}$.

3.3 Model Evaluation Criteria

The performance of the runtime prediction models on the synthetic datasets is evaluated by 10-fold cross-validation, as recommended in the literature [20–22]. Additionally, we evaluate the performance of the best runtime prediction models tested on the real-world datasets. Both the cross-validation folds and the performance on the test set are scored with two error measures, namely Mean Absolute Error (MAE) and the Root Mean Squared Error (RMSE).

Let y_i and $\hat{y}_i$ ($i = 1, ..., n$, with n denoting the number of samples) be the true and the fitted runtime, respectively. The MAE and RMSE measures are defined as follows:

$$\text{MAE}(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i|, \tag{6}$$

$$\text{RMSE}(y, \hat{y}) = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2}, \tag{7}$$

where y and $\hat{y}$ denote all n true and fitted runtimes of one method, respectively. MAE represents the average prediction error and is not prone to outliers. In contrast, RMSE is sensitive to outliers because the error is squared initially [22, 23]. The combination of MAE and RMSE provides information on the origin of the error values. If the MAE and the RMSE are relatively close together, the prediction errors are relatively even in size across the test sets. If the error measure values lie relatively far apart, it means that some runtime prediction errors were much bigger than others.

3.4 Software and Hardware Specifications

For our experiments, we use the six UFS methods available on the ASU Feature Selection Repository [3], and we adapted[4] them to make them suitable for the newer versions of Python and the machine learning package scikit-learn [24]. As mentioned before, these methods belong to the most used and most cited methods in the literature, and they are a good representation of the variety among filter UFS methods (see the taxonomy in Solorio-Fernández *et al.* [6]). We used the default parameter settings given in the ASU Feature Selection Repository for the six methods [3]. Additionally, each method selects 100 features from the given dataset, apart from the Low Variance method. The Low Variance method selects the features with a variance higher than $p(1-p)$, where p is the variance threshold. We used the default setting of $p = 0.1$. The runtime of a method on a dataset is determined by the time it took the methods to return the selected features from the time the data was passed to the method. The timing was done with the time function of the time module in Python.

The experiment was run on the Peregrine compute cluster of the University of Groningen, which made it possible to run the methods on the quantity and dimensions of the datasets as previously described. Moreover, a compute cluster provides the additional advantage of more reliable runtimes because the system is less cluttered by other tasks demanding a machine's resources, such as software updates and antivirus programs. To run something on a compute cluster, one must create a job script to specify what needs to be done. The univariate methods were used to analyze all datasets in one job, which in our case means that the methods analyzed all the datasets consecutively on the same node and CPU. For the slower multivariate methods, we submitted many individual jobs where the methods analyzed one to three datasets at a time, depending on expected and observed runtime. This division of datasets onto many jobs allowed us to get the runtime data in a reasonable period of time.

All these jobs were run on Intel Xeon E5 2680v3 CPUs @ 2.5 GHz. Because not all cores are in use all the time, the clock speed could be as high as 3.3 GHz. Furthermore, the jobs were run with 8 GB of reserved memory. The only exceptions to this are the jobs for UDFS and NDFS, where they analyzed the GLI-85 and the SMK-CAN-187 datasets for which they could use up to 64 GB of memory. 8 GB memory resulted in memory shortage errors. The OS of the Peregrine high-performance cluster when running the experiment was CentOS Linux, release 7.8.2003. The versions of the Python packages are available in the requirements text file on the GitHub page of this paper.

4 Experimental Results

In this section, we present the evaluation of the runtime prediction models. First, we describe the figures and tables. Afterwards, the methods are discussed in the

[4] Further details on the specific requirements and adaptations can be found on the GitHub repository of this paper https://github.com/FeatureSelection/UFS

same order as in Section 2, i.e., Low Variance, Laplacian Score, SPEC, MCFS, UDFS, and NDFS. Each method is discussed based on the observed patterns on Figures 1, 2 and 3, which helped in the design of the models. Afterwards, we assessed the models by using the error criteria presented in Table 3. Finally, we present the final runtime prediction model for each method based on this analysis and test them on the real-world datasets visible in Figure 4 and Tables 4 and 5.

4.1 Description of Figures and Tables

The plots with the runtimes of each method applied on the synthetic datasets are shown in Figures 1, 2 and 3. The position of the points on the vertical axis represents the time a method took to analyze a dataset. Additionally, there are legends encoded by color which represent the category of the points in the figure. Figures 1, 2, and 3 differ in what the x-axis represents. Figure 1 has the number of objects of the analyzed dataset on the x-axis. The number of features of the same dataset is represented by the size of the point with the principle of the bigger the point, the higher the number of features. In Figure 2, the x-axis represents the number of features and the point size represents the number of objects. Lastly, in Figure 3 the x-axis represents the number of data points of a dataset (the product of objects and features).

For Figures 1 and 2 it is important to realize that as the x-axis increases in value, the number of runtime points gradually decreases. This is the effect of creating the synthetic datasets with a maximum of 10,000,000 data points per dataset. As a result, in Figures 1 and 2 the runtime does not seem to increase as quickly as it perhaps should. To combat this potentially misleading representation, the third variable (the number of features or the number of objects) is represented by the size of the data point, as described in the previous paragraph. Figures 1, 2 and 3 also include the fitted runtimes of the best runtime prediction model for each method. These fitted runtimes are only plotted in the figure with the most runtime determining factor as value on the x-axis. The best runtime prediction models will be discussed in Section 4.2. Figure 4 shows the true runtimes of each method applied on the real-world datasets and the predicted runtime by the best model(s). For Low Variance, UDFS and NDFS, more information was needed to pick the best model, so the predictions of those models are both plotted in Figure 4. Notice that the x-axis differs per plot; the independent variable with the most impact on the runtime is presented on the x-axis.

Table 3 shows the mean scores of the 10-fold cross-validation procedure for each combination of method and prediction model for the synthetic datasets. Notice that all but the UDFS and NDFS methods are rounded to three decimals. UDFS and NDFS scores are integers because decimals are unnecessary with numbers over a thousand in our case. Furthermore, the nonlinear model has two scores, one where x_1 = number of objects ($\text{Exp}_{\text{objects}}$) and one where x_1 = number of features ($\text{Exp}_{\text{features}}$) as defined in Equation 5. Table 4 complements Figure 4 by representing the performance of the runtime prediction models. The

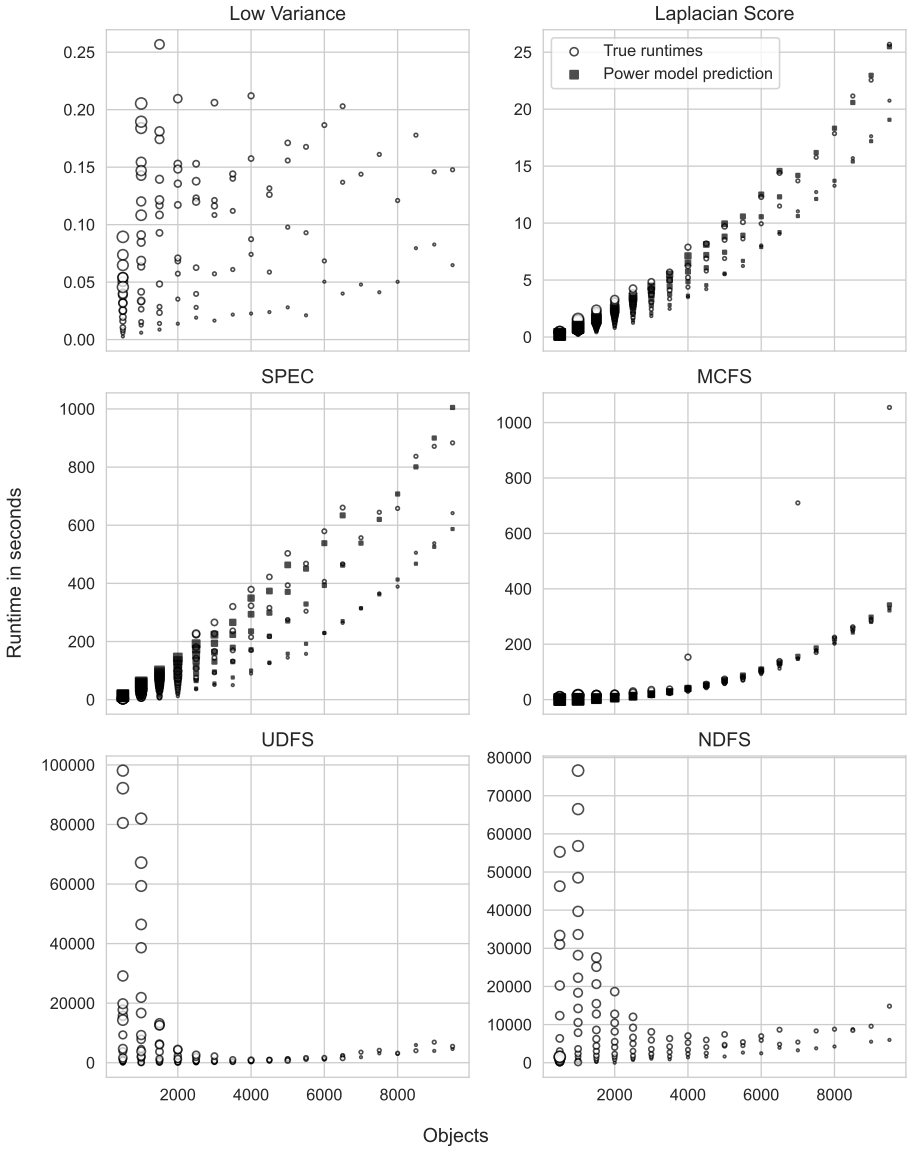

Fig. 1. Runtime results in seconds with objects as x-axis for each method. The legend in one plot describes every plot in the figure. The model that best predicts a method's runtime is also shown. For more details, see Section 4.1.

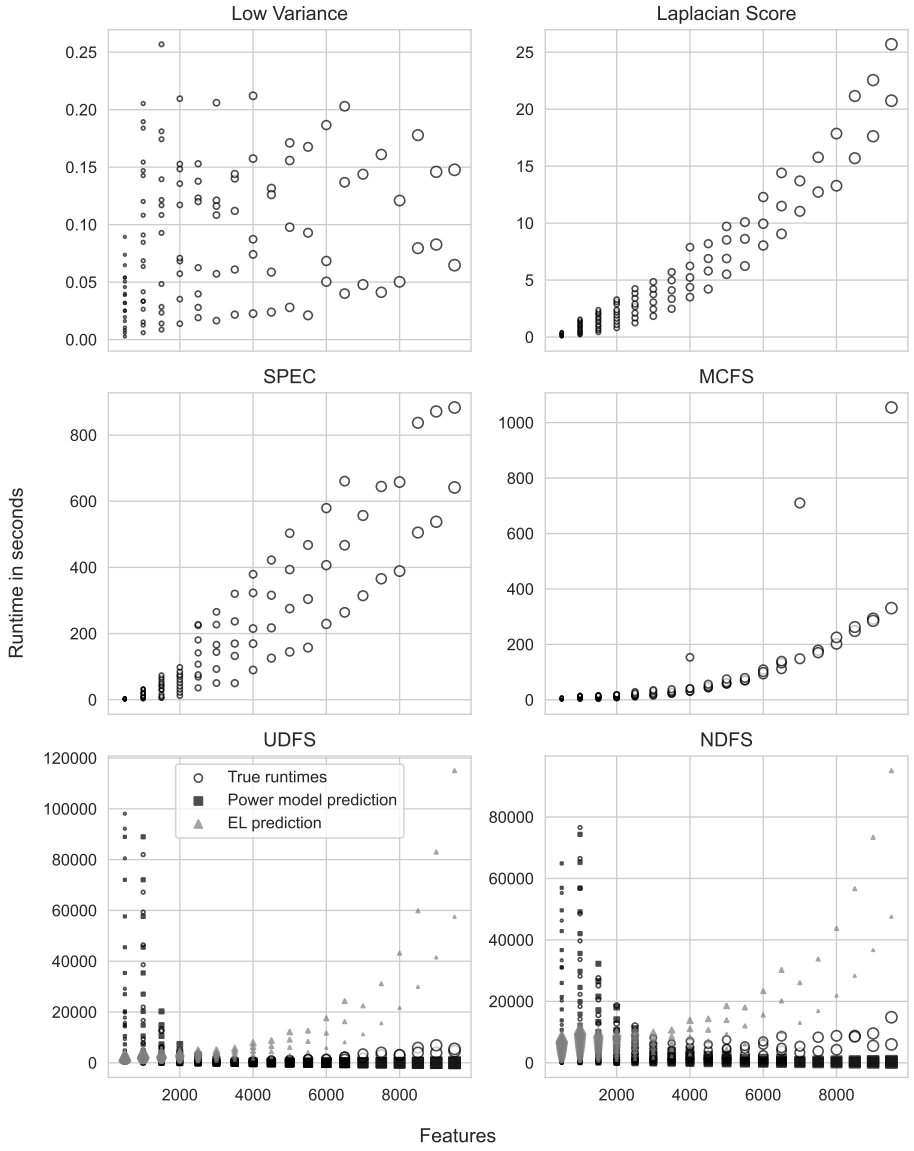

Fig. 2. Runtime results in seconds with features as x-axis for each method. The legend in one plot describes every plot in the figure. The models that best predict a method's runtime are also shown. For more details, see Section 4.1.

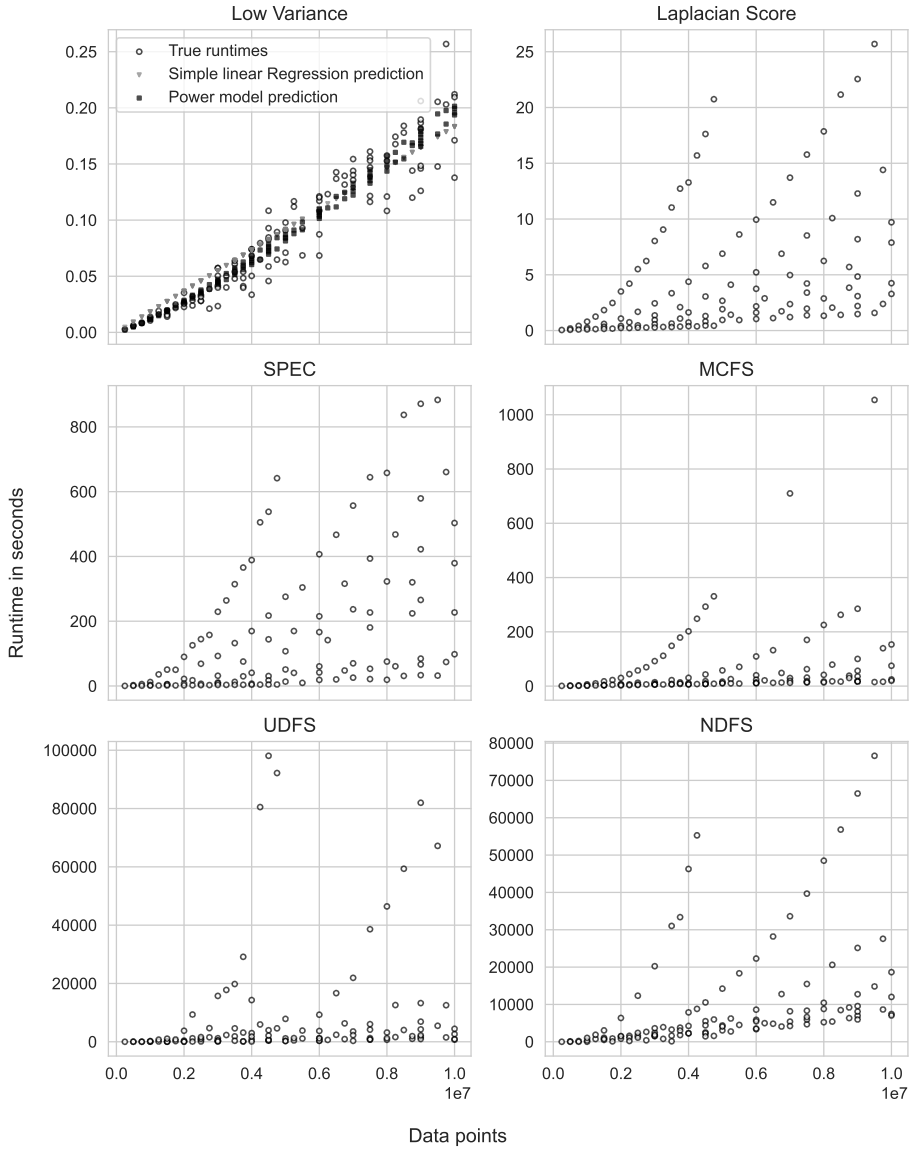

Fig. 3. Runtime results in seconds with objects as x-axis for each method. The legend in one plot describes every plot in the figure. The models that best predict a method's runtime are also shown. For more details, see Section 4.1.

Table 3. Synthetic data error scores for the prediction models in seconds.

Method	Mean Runtime	Measure	Simple linear	Multiple linear	Power	Exponential x_1 =objects	Exponential x_1 =features
Low Variance	0.087	MAE	0.017	0.041	0.009	0.032	0.033
		RMSE	0.021	0.049	0.011	0.038	0.039
Laplacian Score	4.424	MAE	3.625	1.510	0.187	1.295	1.453
		RMSE	4.822	1.960	0.229	1.559	1.761
SPEC	163.221	MAE	139.523	77.251	11.192	56.749	70.658
		RMSE	183.213	94.154	14.095	67.328	83.991
MCFS	43.223	MAE	42.632	26.558	3.940	8.583	15.372
		RMSE	59.515	33.614	4.834	12.782	18.696
UDFS	7997	MAE	10103	9439	1794	6092	2639
		RMSE	15413	12960	2567	7800	4039
NDFS	10244	MAE	8556	5508	1585	4508	1867
		RMSE	12146	7302	2169	5434	2814

table shows the true and predicted runtimes with a high number of objects or features. This allows for a better and zoomed-in representation of the other runtime predictions. Table 5 is similar to Table 3, but it describes the best runtime prediction models applied on the real-world datasets instead of all the runtime prediction models applied on the synthetic datasets.

Notice that these error scores have major flaws in capturing the performance of the runtime prediction models on the real-world test data. This is mostly due to the greatly varying dataset dimensions. There are many smaller datasets and some bigger ones, as visible in Table 2. This distorts the means, and as a result, the error measures do not accurately represent the performance of a model. A better insight can be gained by scrutinizing the plots.

4.2 Runtime Analysis

In the following, we provide a brief description and analysis of the best runtime prediction models for each UFS method applied to the datasets of Tables 1 and 2.

Low Variance For the Low Variance method, the influence of the number of objects and features on the runtime is best represented by having the number of data points on the x-axis, as shown in Figure 3. We see a mostly linear relationship with some variation.

The error scores in Table 3 partially confirm this idea. The MAE and RMSE scores for the simple linear regression model are 0.017 and 0.021, respectively, but the power model has even lower error scores with 0.009 and 0.011. Although the power model has better scores, a linear model might generalize better to datasets with differing dimensions, whereas the power model can be overfitted on the training data.

This is confirmed by testing both models on the real-world data visible in Figure 4. Therefore, the model that best predicts the runtimes of the Low

Variance method in our experiment is:

$$\hat{y} = 1.833 \times 10^{-8} x, \tag{8}$$

where $\hat{y}$ is the predicted runtime and x is the number of data points of a dataset. Figure 4 and the corresponding Table 4 show that the runtimes are predicted well, with the connotation that the predictors are consistently slightly higher than the true runtimes.

Laplacian Score The Laplacian Score plot in Figure 1 shows a nonlinear relation between the number of objects and the runtime. Additionally, it shows that the number of features affects the runtime too, because bigger points have higher runtimes (see Figures 1 and 2). Therefore, we suspect that the power model will perform the best. Our error measures support this, since the error scores for the power model are nearly seven times smaller than the nearest competitor (see Table 3). Therefore, the model that best predicts the runtime of the Laplacian Score method is:

$$\hat{y} = 3.335 \times 10^{-8} x_1^{1.918} x_2^{0.418}, \tag{9}$$

where $\hat{y}$ again denotes the predicted runtime, x_1 denotes the number of objects, and x_2 the number of features of a dataset. Figure 4 and Table 4 show that the power regression model accurately predicts the runtimes of the Laplacian Score method.

SPEC The SPEC plot in Figure 1 shows a similar pattern as the Laplacian Score plot. We see a nonlinear relation between the number of objects and runtimes, and we see that features have a clear influence on the runtime as well. However, there seems to be more variation in runtimes, partially caused by a relatively bigger influence of features on the runtime than with the Laplacian Score method. We hypothesize that the power model will perform the best among the models. This hypothesis is supported by the error measures for which the power model has some five times lower scores than the nearest model, as shown in Table 3. Therefore, the resulting prediction model for the SPEC method is:

$$\hat{y} = 3.507 \times 10^{-8} x_1^{2.044} x_2^{0.776}, \tag{10}$$

with the same definitions as with the Laplacian Score method. Similar to the Laplacian Score method, Figure 4 and Table 4 show that the power regression model accurately predicts the method's runtimes.

MCFS We see in Figure 1 that the MCFS method shows a strong nonlinear relation between objects and runtimes, with a seemingly minimal role of feature numbers (also see Figure 2). Therefore, we hypothesize that the power model suits the data best. Moreover, we see three outliers. We ran the experiment for a second time, and the same three outliers remained. To improve generalization,

we removed these three outliers to fit the models and calculate the MAE and RMSE scores.

As we can observe in Table 3 the error scores provide support for our hypothesis. The scores for the power model are 3.940 and 4.834 for MAE and RMSE, respectively, where the closest contender is the $\text{Exp}_{\text{objects}}$ model with MAE and RMSE scores of 8.583 and 12.782. Thus, the final prediction model for the MCFS method is:

$$\hat{y} = 1.183 \times 10^{-8} x_1^{2.564} x_2^{0.087}, \tag{11}$$

again with $\hat{y}$ denoting the predicted runtime, x_1 denoting the number of objects, and x_2 denoting the number of features of a dataset. In Figure 4 we observe that the runtimes of the MCFS are not predicted well for smaller object sizes, because the model underestimates the effect of the number of features on the runtime. However, the runtimes are predicted with more accuracy when the number of objects increases (see especially Table 4).

UDFS In the UDFS plot in Figure 2 we see a nonlinear relationship between feature numbers and runtimes. The number of objects seems to have a relatively minor effect on the runtime. Lastly, there seems to be relatively much variation as feature sizes increase. These observations lead us to suspect that the power model performs best and that the $\text{Exp}_{\text{features}}$ will perform well too. This suspicion is made more certain by the error measures in Table 3. The scores for the power model are 1794 and 2567 for MAE and RMSE, and the scores for the $\text{Exp}_{\text{features}}$ model are 2639 and 4039. These scores do fit the power model better, but, to make a better substantiated choice, we plotted both models in Figure 4 and show the extra information in Table 4.

In Figure 4 we see that for datasets with relatively low number of features, the runtime predictions are quite accurate. However, when the number of features increases, the prediction quality rapidly decreases. In Table 4 we see that for the datasets with around 20,000 features, the runtime predictions are much larger than the true runtimes. Predictions are especially inaccurate for the $\text{Exp}_{\text{features}}$ model. Therefore, the runtime prediction model for the UDFS method is:

$$\hat{y} = 2.676 \times 10^{-11} x_1^{0.000542} x_2^{3.902}, \tag{12}$$

where the same definitions apply as with the last three models. Notice that in contrast to the last three runtime prediction models, β_1 is much lower than β_2, indicating that the number of features determines the runtime much more than the number of objects. This is in line with the plots we see in Figures 1, 2, 3 and 4.

NDFS Similar to the UDFS method, the NDFS plot in Figure 2 shows a relatively strong nonlinear relationship between feature numbers and runtimes, with a relatively small effect of the number of objects on the runtime. NDFS

Table 4. Measured and predicted runtimes with a high number of objects or features for the real-world datasets (Fig 4).

Method	Dataset	Objects	Features	True Runtime	Model	Predicted Runtime	Methods	Predicted Runtime
Low Variance	Gisette	7000	5000	0.422	Simple linear	0.642	Power	0.914
Laplacian Score	Gisette	7000	5000	41.008	Power	27.808	-	-
Laplacian Score	USPS	9298	256	17.335	Power	13.839	-	-
SPEC	Gisette	7000	5000	2092.370	Power	1882.422	-	-
SPEC	USPS	9298	256	414.148	Power	335.067	-	-
MCFS	Gisette	7000	5000	263.291	Power	179.319	-	-
MCFS	USPS	9298	256	332.579	Power	286.716	-	-
UDFS	SMK-CAN-187	187	19993	157422.626	Power	1,624,588	$\text{Exp}_{\text{features}}$	20,309,840
UDFS	GLI-85	85	22283	484025.000	Power	2,479,277	$\text{Exp}_{\text{features}}$	41,182,695
NDFS	SMK-CAN-187	187	19993	19170.060	Power	322,335	$\text{Exp}_{\text{features}}$	4,223,331
NDFS	GLI-85	85	22283	19163.363	Power	358,746	$\text{Exp}_{\text{features}}$	6,274,883

seems to have less variance than UDFS. Again, we hypothesize that the power model performs best and that the $\text{Exp}_{\text{features}}$ model will perform well too.

Similar to MCFS, for NDFS, some outliers were produced. We did not consider these outliers for fitting the models and calculating the error scores, as we did with the outliers in MCFS. Notice that these outliers are not too problematic because they finish much quicker than regular predicted runtimes. The number of outliers is noteworthy, however, with 7 out of 114 runtimes classified as outlier. Running the experiment for a second time resulted in the same outliers.

The error measures in Table 3 share the observation of the power model and the $\text{Exp}_{\text{features}}$ model performing best. The scores for the power model are 1585 and 2169 for MAE and RMSE, and the scores for the $\text{Exp}_{\text{features}}$ model are 1867 and 2814. These scores do fit the power model better, but to further investigate, we plotted both results of the runtime prediction models in Figure 4 and present the extra information in Table 4.

Figure 4 and Table 4 show that both the power model and the $\text{Exp}_{\text{features}}$ model do not predict the runtimes well. Table 4 shows that while the true runtimes for two high dimensional datasets are around 19,000 seconds, whereas the power predicts around 322,000 and 358,000 seconds respectively, and the $\text{Exp}_{\text{features}}$ model predicts around 4,000,000 and 6,000,000, respectively. Therefore, the power model predicts the runtimes best for the NDFS method, with the model being:

$$\hat{y} = 4.890 \times 10^{-6} \times x_1^{0.196} x_2^{2.412} \tag{13}$$

where the same definitions and remarks apply as with the UDFS method.

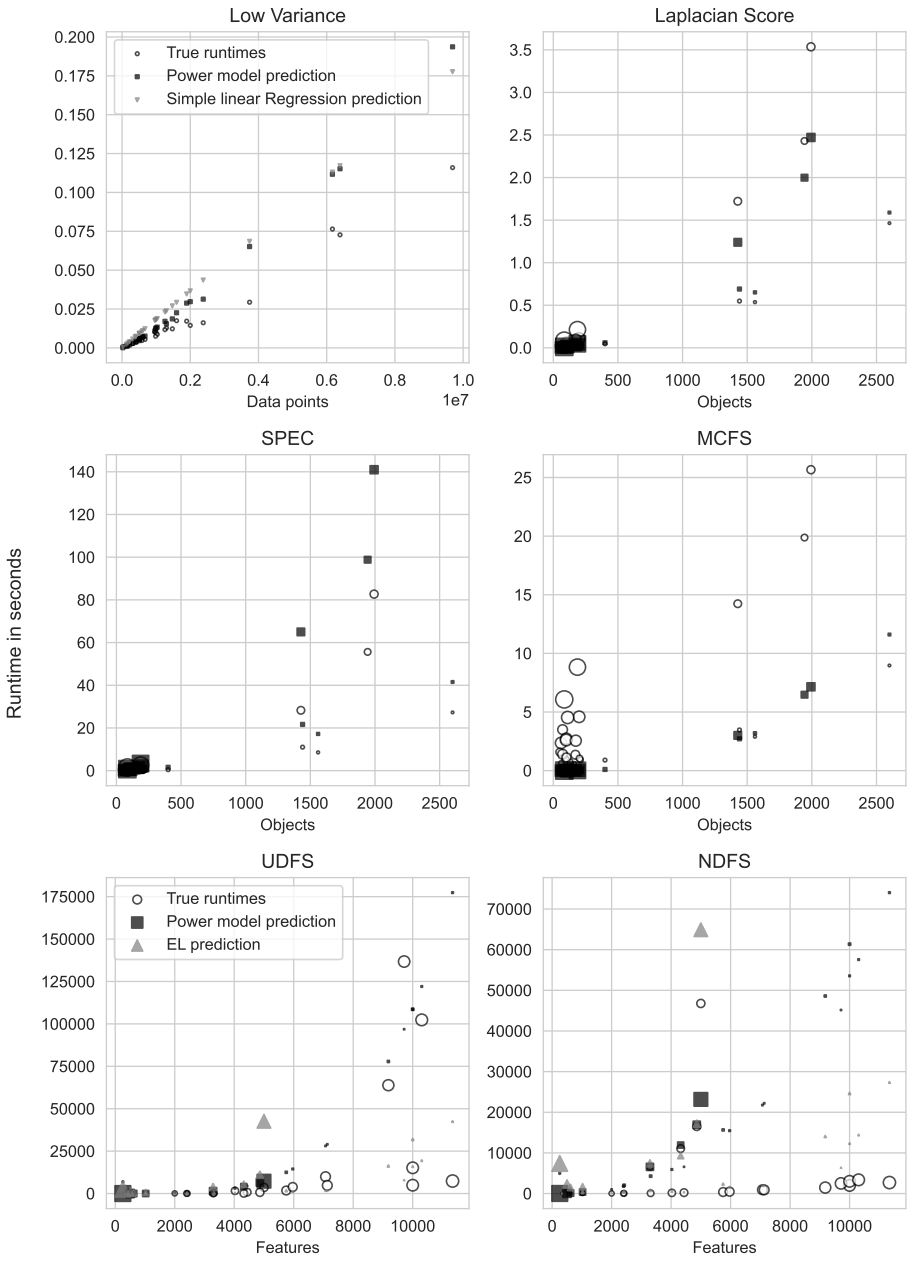

Fig. 4. Runtime results in seconds with objects as x-axis for each method. The top legend describes the top four plots, and the bottom legend the bottom two. The models that best predict a method's runtime are shown as well. For more details, see Section 4.1.

Table 5. The error scores for the best prediction models in seconds.

Method	Mean Runtime	Best model	Error	Score	Contender Model	Error	Scores
Low Variance	0.018	Simple linear	MAE RMSE	0.027 0.009	Power	MAE RMSE	0.021 0.002
Laplacian Score	0.473	Power	MAE RMSE	0.682 6.480	-	MAE RMSE	- -
SPEC	9.656	Power	MAE RMSE	16.360 2055.675	-	MAE RMSE	- -
MCFS	4.487	Power	MAE RMSE	7.771 340.712	-	MAE RMSE	- -
UDFS	10219	Power	MAE RMSE	137506 2.122×10^{11}	$\text{Exp}_{\text{features}}$	MAE RMSE	2118501.956 7.155×10^{13}
NDFS	3950	Power	MAE RMSE	37653.890 7.822×10^{9}	$\text{Exp}_{\text{features}}$	MAE RMSE	349876.063 1.800×10^{12}

5 Discussion

In this section, we first discuss the performance of the runtime prediction models, followed by the section on general runtime observations. Then, we address related literature. Finally, we discuss some limitations of our research.

5.1 Performance of Runtime Prediction Models

The runtime prediction models performed better for the Low Variance, Laplacian Score, SPEC, and MCFS methods than for the UDFS and NDFS methods. The low error scores and near predictions presented in Tables 3, 4 and 5 and in Figures 1, 2, 3 and 4 illustrated the quality of prediction models for the four methods. Especially relevant are Figure 4 and the associated Table 4, which indicate how well the models trained on the synthetic datasets generalize to the real-world datasets. Clearly, the runtime predictions are not perfectly accurate. However, this was not the goal of this paper. More importantly, these predictions can give users a priori runtime information of a method. Additionally, these predictions expose the influence of the number of objects and features on the runtime, which will be discussed in Section 5.2.

The performance of the runtime prediction models for the UDFS and NDFS methods is worse than for the other four methods. Although the power model predicts the runtimes of the synthetic datasets well, the prediction performance on some real-world datasets decreases, particularly when the number of features is high. In general, our four regression models fail to capture the influence of the number of objects and features of a dataset on the runtime for both methods. More specifically, we believe that the runtime prediction models for UDFS and NDFS have low performance because of the following observation. When the number of objects increases and the number of features is fixed, we see a reasonably clear nonlinear relationship between an increase in the number of

objects and the increase in the runtime. It is likely that our models would be able to capture this interaction. However, this interaction depends on the number of features too. For example, the difference in runtime between dataset A (2000, 1000) and dataset B (2500, 1000) is different between the runtime differences in dataset C (2000, 2000) and dataset D (2500, 2000). Although the increase of object numbers between A & B are the same as between C & D, the runtime will not increase with the same amount (even while ignoring runtime differences caused by the environment itself). Of course, we see a similar phenomenon when the roles of objects and features are switched. Our models likely failed to capture these important and more complex interactions. This deficiency becomes evident when the number of objects and the number of features of a dataset greatly differ from the synthetic training set, which is the case for some real-world datasets. The prediction errors we see in Table 4 are likely the result of this deficiency.

5.2 Runtime Observations

The runtime prediction results are generally in line with the theory on univariate and multivariate methods, which would suggest that the number of features is less important for univariate methods than for multivariate methods. Whereas univariate methods analyze the features separately, multivariate methods aim to find an optimal subset of features. This requires solving the "subset sum" problem that gets increasingly difficult as the whole set of features increases in size. Indeed, we observed that the number of features affects the runtimes most for the UDFS and NDFS methods, which is not the case for the univariate methods. An exception to this is the multivariate MCFS method, where the number of objects has a stronger influence on the runtime than the number of features. Furthermore, we clearly see differences in the order of runtimes.

The Low Variance and the Laplacian Score methods are the quickest methods and can be employed to analyze large datasets. The Laplacian Score method is particularly quick in analyzing highly dimensional datasets where the number of objects is relatively low and the number of features is much higher. On the other hand, the SPEC and MCFS methods are clearly slower than the previous two methods, but in our experiment, they operated in the order of seconds and minutes for larger datasets. Consequently, they can typically execute within most time constraints of research projects. The SPEC method is best applied when the number of objects of a dataset is not too high, while the number of features can be large. The same holds for the MCFS method, although the number of features has a relatively higher effect on the runtimes than with the SPEC method.

Lastly, the UDFS and NDFS methods are the slowest methods. Although our models have low performance in predicting their runtimes, the runtime data is still useful to provide runtime guidelines. In our environment, both UDFS and NDFS take a couple of days to complete when the number of features exceeds 10,000. We have seen that the number of features has a strong linear effect on the runtime; thus, the runtimes of these methods might easily take weeks and months when the number of features exceeds 10,000. Do notice that this is based only on datasets with around 100 objects. It is unknown what runtimes to expect

with different object sizes. Nonetheless, long runtimes should be considered when these methods are applied on high-dimensional datasets.

5.3 Related Literature

As far as we know, this experiment is unique in the field of Unsupervised Feature Selection. However, method runtimes have been examined in other research fields, especially in optimizing job scheduling in high-performance clusters [22, 25]. Random forest regression models often perform well in these fields because they can take many independent variables into account. However, we focus on two independent variables in our study. Moreover, random forest has difficulties with extrapolating from training data, which in our case means that the runtime predictions will not be accurate when datasets have considerably different dimensions than the dimensions of synthetic training datasets [22]. Unfortunately, we cannot train our models on all the potential dataset dimensions users might have because of these two reasons; therefore, the random forest algorithm is not suitable for this study.

On the other hand, we used a benchmarking approach in this paper, but another possible approach was to focus even more on a mathematical analysis of methods. The Big O notation is often used to represent the time and space complexity of a method [26]. The Big O notation describes the general computational operations that a method performs, but it ignores important factors for runtime analysis, such as the machine, the programming language, and the compiler the method runs in. Moreover, the time complexity is only available for two out of the six methods, namely the SPEC and MCFS methods. Still, time complexity analysis such as in Cai *et al.* [13] and Zhao & Liu [27] can be useful to examine the interaction of object and feature numbers on the runtime. In this study, we did use the time complexity of the SPEC and the MCFS method to create runtime prediction models.

Finally, it is noteworthy that the runtime prediction for the UDFS and NDFS methods can probably best be improved by analyzing the time complexity of the methods by examining the original paper where the methods are presented (Yang *et al.* [17] and Li *et al.* [18], respectively). However, the time complexity has not been provided by the authors themselves and analyzing their time complexity was out of the scope of this project.

5.4 Limitations

Our experiment is also subject to some limitations. Most notably, as we stated in Section 3.2, generalizing the runtime findings from our experiment environment to a user's environment brings along complications. Although this was not part of our research objective, it does interfere with the extrapolation of our findings to the environment of a user. On the other hand, a missed insight that is relevant to our research goal are the outliers of the MCFS and NDFS methods, visible in Figures 1 and 2. It is unknown to us why exactly these outliers exist and the effect it has on the clustering and classification performance. For the NDFS method,

the outliers are less likely to be problematic, as the method analyzes much faster than expected. The outliers for the MCFS methods can be problematic as the runtime is multiple factors higher than expected.

Additionally, the models and their parameters do not necessarily represent an optimal fit. It could be that other simple models, such as polynomial ones, fit the runtime data better. As for the model parameters, Trust Region Reflective algorithm, used to optimize the parameters in the nonlinear models, does not converge to a global minimum [19]. Consequently, it could be the case that with different initial parameter guesses, the models would have better-fitted parameters. Furthermore, the effect of the hypercube size multiplication factor, used in generating the synthetic datasets, on the runtime is unknown. It could be that a higher factor speeds up some methods when, for example, pseudo-class labels are learned with spectral analysis in the NDFS method.

Lastly, the effects of method parameters, such as the number of selected features and the number of nearest neighbours, used in the MCFS method, for example, are left unstudied.

6 Concluding Remarks and Future Work

In this study, we have presented runtime prediction models for six relevant and classical filter UFS methods of the state-of-the-art. The runtime prediction models and the general guidelines for each of the six methods can be particularly useful for professionals and practitioners in this research field. Moreover, our results, in line with previous work on the evaluation of filter UFS methods [7], could be useful to assist users in choosing an appropriate method for a particular problem. From the results presented in the previous sections and the analysis performed, we contribute to the runtime knowledge of filter UFS methods by providing some insights and guidelines:

- The Low Variance, Laplacian Score, SPEC, and MCFS methods are much faster than the UDFS and NDFS methods.
- The Low Variance method is the quickest method, which can be applied on large datasets in most cases without runtime problems. The runtime is best determined by the number of data points of a dataset.
- The Laplacian Score method can be applied on large datasets as well, and is especially efficient in analyzing high-dimensional datasets.
- The SPEC method is considerably slower than the previous two methods, but its runtimes will still often be manageable. The SPEC method is best at analyzing high-dimensional datasets. However, a large number of objects increase the runtime considerably.
- The MCFS method is the fastest multivariate method, even faster than the univariate SPEC method. Surprisingly, the runtimes of the MCFS method are most influenced by the number of objects, meaning that it handles high-dimensional datasets well.

- The UDFS method is substantially slower than the previous four methods. UDFS should be used carefully with large datasets, especially when datasets have many features (roughly $> 10,000$).
- The NDFS method has similar runtimes and should be used with the same care as the UDFS method.

Finally, future work of this research includes the following:

- Analyzing the time complexity of the UDFS and NDFS methods and building corresponding runtime prediction models to improve the current predictions.
- Performing experiments on datasets with different shapes to generalize and improve the general performance of the runtime prediction models.
- Investigating runtime prediction in other environments could improve the usability of our research. Future research similar to, or in combination with, a paper by Sidnev [28] might be fruitful.
- A similar study like ours can be used to examine the runtimes of other filter methods, and it can be extended to wrapper and hybrid (embedded) UFS methods as well.

Acknowledgments. We want to thank the Center for Information Technology of the University of Groningen for their support and for providing access to the Peregrine high-performance computing cluster.

References

1. Iguyon, I. & Elisseeff, A. *An introduction to variable and feature selection* 2003.
2. Bellman, R. Combinatorial processes and dynamic programming. https://apps.dtic.mil/sti/pdfs/AD0606844.pdf (1958).
3. Li, J. *et al. Feature selection: A data perspective* 2017. arXiv: 1601.07996. https://doi.org/10.1145/3136625.
4. Alelyani, S., Tang, J. & Liu, H. *Feature selection for clustering: a review* (eds Aggarwal, C. C. & Reddy, C. K.) 29–60. ISBN: 9781728128474 (Taylor & Francis, 2014).
5. Liu, L., Kang, J., Yu, J. & Wang, Z. A comparative study on unsupervised feature selection methods for text clustering. *Proceedings of 2005 IEEE International Conference on Natural Language Processing and Knowledge Engineering, IEEE NLP-KE'05* **2005,** 597–601 (2005).
6. Solorio-Fernández, S., Carrasco-Ochoa, J. A. & Martínez-Trinidad, J. F. A review of unsupervised feature selection methods. *Artificial Intelligence Review* **53,** 907–948. ISSN: 15737462. https://doi.org/10.1007/s10462-019-09682-y (2020).
7. Solorio-Fernández, S., Carrasco-Ochoa, J. A. & Martínez-Trinidad, J. F. A systematic evaluation of filter Unsupervised Feature Selection methods. *Expert Systems with Applications* **162.** ISSN: 09574174 (2020).

8. Fredriksson, T., Mattos, D. I., Bosch, J. & Olsson, H. H. *Data Labeling: An Empirical Investigation into Industrial Challenges and Mitigation Strategies* in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* **12562 LNCS** (Springer Science and Business Media Deutschland GmbH, 2020), 202–216. ISBN: 9783030641474. https://doi.org/10.1007/978-3-030-64148-1%7B%5C_%7D13.
9. Dong, G. & Liu, H. *Feature engineering for machine learning and data analytics* 192–194. ISBN: 9781138744387 (Taylor & Francis, 2018).
10. Goldberg, Y. & Orwant, J. A Dataset of Syntactic-Ngrams over Time from a Very Large Corpus of English Books. **SEM 2013 - 2nd Joint Conference on Lexical and Computational Semantics* **1,** 241–247 (2013).
11. Harper, F. M. & Konstan, J. A. The movielens datasets: History and context. *ACM Transactions on Interactive Intelligent Systems* **5,** 1–19. ISSN: 21606463 (2015).
12. Bolón-Canedo, V., Sánchez-Maroño, N. & Alonso-Betanzos, A. *Artificial Intelligence: Foundations, Theory, and Algorithms Feature Selection for High-Dimensional Data - Chapter 3: A Critical Review of Feature Selection Methods* 1–10. ISBN: 9783319218571. www.springer.com/series/ (Springer, 2016).
13. Cai, D., Zhang, C. & He, X. Unsupervised feature selection for Multi-Cluster data. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining,* 333–342 (2010).
14. Dy, J. G. & Brodley, C. E. *Feature Selection for Unsupervised Learning* tech. rep. (2004), 845–889.
15. He, X., Cai, D. & Niyogi, P. Laplacian Score for feature selection. *Advances in Neural Information Processing Systems,* 507–514. ISSN: 10495258 (2005).
16. John, G. H., Kohavi, R. & Pfleger, K. Irrelevant Features and the Subset Selection Problem. *Machine Learning Proceedings 1994,* 121–129 (1994).
17. Yang, Y., Shen, H. T., Ma, Z., Huang, Z. & Zhou, X. l2,1-Norm regularized discriminative feature selection for unsupervised learning. *IJCAI International Joint Conference on Artificial Intelligence,* 1589–1594. ISSN: 10450823 (2011).
18. Li, Z., Yang, Y., Liu, J., Zhou, X. & Lu, H. Unsupervised feature selection using nonnegative spectral analysis. *Proceedings of the National Conference on Artificial Intelligence* **2,** 1026–1032 (2012).
19. Branch, M. A., Coleman, T. F. & Li, Y. Subspace, interior, and conjugate gradient method for large-scale bound-constrained minimization problems. *SIAM Journal of Scientific Computing* **21,** 1–23. ISSN: 10648275 (1999).
20. Hastie, T., Tibshirani, R. & Friedman, J. The Elements of Statistical Learning. *The Mathematical Intelligencer* **27,** 241–247. ISSN: 03436993 (2017).
21. Arlot, S. & Celisse, A. A survey of cross-validation procedures for model selection. *Statistics Surveys* **4,** 40–79. ISSN: 19357516. arXiv: 0907.4728 (2010).

22. Hutter, F., Xu, L., Hoos, H. H. & Leyton-Brown, K. Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence* **206,** 79–111. ISSN: 00043702. www.elsevier.com/locate/artint (2014).
23. Chai, T. & Draxler, R. R. Root mean square error (RMSE) or mean absolute error (MAE)? -Arguments against avoiding RMSE in the literature. *Geoscientific Model Development* **7,** 1247–1250. ISSN: 19919603 (2014).
24. Pedregosa, F. *et al.* Scikit-learn: Machine Learning in Python. *Machine Learning Research* **12,** 2825–2830 (2011).
25. McKenna, R., Herbein, S., Moody, A., Gamblin, T. & Taufer, M. Machine learning predictions of runtime and IO traffic on high-end clusters. *Proceedings - IEEE International Conference on Cluster Computing, ICCC,* 255–258. ISSN: 15525244 (2016).
26. Russel, S. & Norvig, P. *Artificial Intelligence A Modern Approach (4th Edition)* **9.** ISBN: 9788578110796 (Pearson, 2020).
27. Zhao, Z. & Liu, H. Spectral feature selection for supervised and unsupervised learning. *ACM International Conference Proceeding Series* **227,** 1151–1157 (2007).
28. Sidnev, A. A. Runtime prediction on new architectures. *ACM International Conference Proceeding Series* **23-24-Octo,** 1–6 (2014).