Your Report Title

Your Name

October 27, 2024

1 Introduction

This is the introduction section of the report. It provides background information and context for the study, states the research question or hypothesis, and briefly outlines the approach taken. A report structure overview is also provided to guide the reader through the document.

1.1 Background Information

Classification in the context of machine learning is the process of predicting the class or category of a given data point based on its features. For example: given a set of emails, classify each email as spam or not spam. It is a supervised learning technique that is used to assign labels to data points based on their characteristics. Classification is an ideal technique for the required analysis as outlined in this report.

K-Nearest Neighbour (k-NN) is a supervised machine learning algorithm that has been chosen as the primary method for this study. It is a simple yet powerful algorithm that classifies new cases based on similarity to existing data points. While k-NN can be used for both classification and regression problems, our focus is on classification due to the nature of the datasets used.

This report details the analysis of two datasets: 'hepatitis' and 'mushroom'. These are two of many datasets provided to us as part of this assignment. We decided upon these two datasets due to the stark contrast between them in terms of complexity and size.

1.2 Research Question or Hypothesis

The primary research question addressed in this report is: 'Can the k-Nearest Neighbour algorithm effectively classify data points in the hepatitis and mushroom datasets with high accuracy'?.

The specific objectives of this study are:

1. To evaluate k-NN's classification performance on datasets of varying sizes and complexities
2. To determine optimal k-NN hyperparameter settings for each dataset
3. To assess the algorithm's robustness and limitations in different scenarios

We hypothesize that k-NN will perform excellently on the mushroom dataset due to the simplicity of the data, but may struggle with the hepatitis dataset due to the complexity of the data in that dataset.

1.3 Approach and Methodology

The approach taken in this report involves the following steps:

1. Data Preprocessing: Both datasets are preprocessed, ensuring the data is clean and ready for analysis.
2. Model Training: Different combinations of hyperparameters are used to train the k-NN algorithm on the training dataset.
3. Model Evaluation: The performance of the best k-NN algorithm from the previous step is evaluated using the test dataset.

1.4 Report Structure

The report is structured as follows:

1. Introduction (Section 1): Provides background information, states the research question, and outlines the approach.
2. Data (Section 2): Describes the data used in the study, along with the source, characteristics, relevance, and the preprocessing techniques that were applied.
3. Methods (Section 3): Describes the methodology used in the study, including the algorithms, techniques, and tools used.
4. Results and Analysis (Section 4): Findings of the study are presented, including relevant data, statistics, and figures to help visualise the data.
5. Conclusion (Section 5): Summarizes main findings of the study and their significance.

2 Data

2.1 Dataset

This section describes the dataset used in the study, including its source, characteristics, and relevance to the research question.

2.1.1 Dataset Overview

2.1.2 Data Collection

2.1.3 Data Characteristics

2.2 Data Preprocessing

This section outlines the steps taken to prepare the dataset for analysis, including cleaning, transformation, and feature engineering.

2.2.1 Data Cleaning

2.2.2 Data Transformation

2.2.3 Feature Engineering

3 Methods

This section describes the methodology used in the study. It should include details about data collection, experimental design, and analytical techniques.

3.1 k-Nearest Neighbors (kNN)

This section describes the k-Nearest Neighbors (kNN) algorithm and its implementation in our study.

3.1.1 Algorithm Overview

The k-NN algorithm operates on a simple yet effective principle: when classifying new data points, it examines the k closest training examples and assigns the most common class among these neighbors (where k is a user-defined hyperparameter). The algorithm's effectiveness relies on two fundamental assumptions:

- Locality: Points that are close to each other are likely to have the same class.
- Smoothness: The classification boundary between classes is relatively smooth.

One key feature of k-NN is it employs neighbor-based classification, where the classification of a new data point is determined by majority voting among its k-nearest neighbors. The value of k is one of the most important tunable hyperparameters, as it significantly influences the algorithm's behaviour:

- Small k values (e.g., k=1 or k=3): More sensitive to local patterns but susceptible to noise.
- Large k values: More robust to noise but may overlook important local patterns.
- Even vs. Odd k values: Even k values result in ties, which may require additional rules to break.

The dependent variable in our datasets we aim to predict is categorical, so while k-NN can be used for both classification and regression problems our focus is on classification.

While the k-NN algorithm has it's merits in terms of simplicity and interpretability, it also has several disadvantages:

- Computationally expensive: As the number of training examples grows, the algorithm's complexity increases[1].
- Sensitive to irrelevant features: The algorithm treats all features equally, so irrelevant features can negatively impact performance.
- Curse of dimensionality: As the number of features increases, the algorithm requires more data to maintain performance.

All three of the above mentioned disadvantages all relate to the features of the dataset, and how they can impact the performance of the algorithm. They create a compounding effect: more features lead to higher computational cost, while making the algorithm more susceptible to noise and irrelevant features, and also requiring more data to maintain performance. This is why the use of feature selection and reduction techniques are imperative when working with the k-NN algorithm to increase performance.

3.1.2 Implementation Details

The K-Nearest Neighbors (k-NN) algorithm is implemented using Python with various libraries and tools. Below are the specific implementation details:

Distance calculations

The Euclidean distance is used to measure the distance between data points. This distance metric is the most commonly used distance metric in k-NN algorithms due to its simplicity and effectiveness in measuring the similarity between data points[2]. It operates on the principle of calculating the straight-line distance between two points in a Euclidean space, hence it's simplicity.

The formula for Euclidean distance between two vectors x and y is:

$$d = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

where x_i and y_i are the ith elements of vectors x and y, respectively.

The Manhattan distance is another distance metric that can be used in k-NN algorithms. It is also frequently used with the k-NN algorithm, albeit not as common as the Euclidean distance. The Manhattan distance is calculated by summing the absolute differences between the coordinates of two points.

The formula for Manhattan distance between two vectors x and y is:

$$d = \sum_{i=1}^{n}|x_i - y_i|$$

where x_i and y_i are the ith elements of vectors x and y, respectively.

The Chebychev distance less commonly used than the Euclidean and Manhattan distances in relation to the k-NN algorithm, but it is still a valid distance metric and operates effectively in measuring the similarity between data points. The Chebychev distance is calculated by taking the maximum absolute difference between the coordinates of two points. This differs from Manhattan distance in that it takes the maximum absolute difference, rather than the sum.

The formula for Chebychev distance between two vectors x and y is:

$$d = \max_{i=1}^{n} |x_i - y_i|$$

where x_i and y_i are the ith elements of vectors x and y, respectively.

Along with the Chebychev distance, the Malahanobis distance is another distance metric used in k-NN algorithms. It is also less commonly used than the Euclidean and Manhattan distance metrics. The Mahalanobis distance is calculated by taking the square root of the sum of the squared differences between the coordinates of two points, where the squared differences are divided by the covariance matrix of the data.

The formula for Mahalanobis distance between two vectors x and y is:

$$d = \sqrt{(x - y)^T \cdot S^{-1} \cdot (x - y)}$$

where x and y are the vectors, and S is the covariance matrix of the data.

Weighting Schemes

In the k-NN algorithm, the choice of weighting scheme can significantly impact the classification results. The following weighting schemes were implemented in this study:

In uniform weighting, all neighbours have equal weight in the voting process. This is the default weighting scheme in k-NN. An advantage of uniform weighting is that it is simple and computationally efficient, but it may not be optimal for imbalanced datasets.

With ReliefF weighting, neighbours are weighted based on their relevance to the target class. This provides a more nuanced approach to weighting as it considers the importance of each neighbour in the classification process, potentially improving performance of the algorithm.

Information gain weighting, Neighbours are weighted based on gain in information in the context of the target variable [3]. Similarly to ReliefF weighting, this weighting scheme assigns higher weights to neighbours that provide more information about the target class.

Voting Schemes

In the k-NN algorithm, the voting scheme determines how the class label of a new data point is determined based on the class labels of its k-nearest neighbors. The following voting schemes were implemented in this study:

In the majority voting scheme, the class label with the highest frequency among the k-nearest neighbors is assigned to the new data point. As well as this, each vote is given equal weight in the voting process [4]. This is the default voting scheme in k-NN and is simple and easy to implement.

With inverse distance weighting voting, the class labels of the k-nearest neighbors are weighted based on their distance from the new data point. While there are different methods to perform this weighting, the most simple version is to take a neighbour's vote to be the inverse of its distance to q:

$$w_i = \frac{1}{d(q, x_i)}$$

where w_i is the weight of the ith neighbour, $d(q, x_i)$ is the distance between the new data point q and the ith neighbour x_i.

Then the votes are summed and the class with the highest votes is returned [4].

Shepard's method is another voting scheme that can be used in k-NN algorithms. It employs the use of an exponential function to weight the votes of the neighbours based on their distance from the new data point, rather than the inverse of the distance [5].

The formula for Shepard's voting scheme is:

$$Vote(y_j) = \sum_{i=1}^{k} e^{-d(\mathbf{q},\mathbf{x}_i)^2} 1(y_j, y_c) \tag{1}$$

where $Vote(y_j)$ is the vote for class y_j, $d(\mathbf{q}, \mathbf{x}_i)$ is the distance between the new data point $\mathbf{q}$ and the ith neighbour $\mathbf{x}_i$, and $1(y_j, y_c)$ is the indicator function that returns 1 if y_j is the same as the class label y_c of the ith neighbour, and 0 otherwise.

Neighbor Selection

The choice of the number of neighbors (k) is a critical hyperparameter in the k-NN algorithm, as previously discussed in this report. Different values of k can significantly impact the algorithm's performance, with smaller values being more sensitive to noise and larger values potentially overlooking important local patterns. It's imperative to choose an optimal value of k that balances these trade-offs and maximizes the algorithm's performance. In this study, the following values of k where examined: [1, 3, 5, 7]

Basic Algorithm Steps

The k-NN algorithm can be summarized in the following steps:

1. Preprocess the data:
    - Scale/normalize features to ensure equal contribution
    - Handle missing values
    - Encode categorical variables if necessary
2. Optimize the model:
    - Tune hyperparameters (k, distance metric, weighting scheme, voting scheme)
    - Consider dimensionality reduction techniques
    - Implement feature selection if necessary
    - Balance dataset if required
3. For each query point $\mathbf{q}$:
    - Calculate distances $d(\mathbf{q}, \mathbf{x}_i)$ to all training examples
    - Sort distances to identify the k-nearest neighbors
    - Apply selected weighting scheme to neighbour votes
    - Determine class label using chosen voting method
4. Validate the model:
    - Split data into training and validation sets
    - Evaluate using appropriate metrics (accuracy, precision, recall, F1)
    - Perform cross-validation to assess generalization

Libraries and Tools

The following libraries and tools were used for the implementation of our study:

- **Python**: The primary programming language used for the implementation of the k-NN algorithm.
- **NumPy**: A useful package for scientific computing with Python, used for numerical operations.
- **Scikit-learn**: A machine learning library in Python, used for implementing the k-NN algorithm.
- **Pandas**: A data manipulation library in Python, used for data preprocessing and analysis.
- **Matplotlib**: A plotting library in Python, used for data visualization.
- **Seaborn**: A data visualization library in Python, used for creating informative and attractive statistical graphics.
- **Jupyter Notebook**: An interactive development environment used for running Python code and visualizing results.
- **SciPy**: A scientific computing library in Python, used for scientific and technical computing.
- **TensorFlow / PyTorch**: An open-source machine learning library in Python, used for building and training machine learning models.

3.1.3 Parameter Tuning

The k-NN algorithm's performance depends significantly on the careful tuning of several key parameters. In this study, we employed a systematic approach to parameter optimization using manual grid search with cross-validation. The following parameters were tuned:

Data Preprocessing

Before parameter tuning, comprehensive preprocessing pipelines were implemented for both the hepatitis and mushroom datasets using scikit-learn's ColumnTransformer and Pipeline classes. The preprocessing steps were customized for each dataset's specific characteristics:

Hepatitis Dataset Preprocessing The hepatitis dataset required handling of both numeric and categorical features:

- **Numeric Features**: The following features were processed using mean imputation and Min-Max scaling:
  - AGE
  - ALK_PHOSPHATE
  - SGOT
  - BILIRUBIN
  - ALBUMIN
  - PROTIME
- **Categorical Features**: The following features were processed using mode imputation and label encoding:
  - SEX
  - STEROID
  - ANTIVIRALS
  - FATIGUE
  - MALAISE
  - ANOREXIA
  - Other binary indicators (LIVER_BIG, LIVER_FIRM, etc.)

Mushroom Dataset Preprocessing The mushroom dataset consisted entirely of categorical features:

- **22 Categorical Features**: Including:
  - cap-shape
  - cap-surface
  - cap-color
  - bruises?
  - odor
  - gill-related features
  - stalk-related features
  - Other morphological characteristics

```
preprocessor = ColumnTransformer(
transformers=[
('num', Pipeline(steps=[
('imputer', SimpleImputer(strategy='mean')),
('scaler', MinMaxScaler())
]), numeric_cols),
('cat', Pipeline(steps=[
('imputer', SimpleImputer(strategy='most_frequent')),
('passthrough', 'passthrough')
]), categorical_cols)
]
)
```

Missing Value Handling

- Question marks ('?') were replaced with `np.nan`
- Numeric features: Missing values imputed using mean strategy
- Categorical features: Missing values imputed using mode strategy

Feature Encoding Label encoding was applied to all categorical features:

```
for col in categorical_cols:
le = LabelEncoder()
processed_df[col] = le.fit_transform(processed_df[col])
```

K Value Selection

The optimal value of k was determined through k-fold cross-validation (k=10) across the candidate k values [1, 3, 5, 7]. For each dataset, we evaluated the performance metrics (accuracy, precision, recall, and F1-score) across these k values. To avoid ties in classification, we primarily focused on odd values of k. The final k value was selected based on the best average performance across all folds, also taking into account the different hyperparameters and their impact on the model's performance.

Distance Metric Optimization

We evaluated four distance metrics:

- Euclidean distance
- Manhattan distance

- Chebyshev distance
- Mahalanobis distance

Each distance metric was tested in combination with different k values to identify the optimal pairing. The Mahalanobis distance required additional computation of the covariance matrix

Weighting Scheme Selection

Three weighting schemes were evaluated through cross-validation:

- Uniform weighting (baseline)
- ReliefF weighting
- Information gain weighting

The optimal weighting scheme was selected based on both performance metrics and computational efficiency considerations.

Voting Scheme Optimization

We compared three voting schemes:

- Majority voting
- Inverse distance weighting
- Shepard's method

Each voting scheme was evaluated across different combinations of k values and distance metrics

Parameter Search Implementation

The optimal combination of parameters was determined using a custom grid search function that evaluated all possible combinations of:

- k values
- Distance functions
- Voting schemes
- Weighting schemes

The final model was selected based on the best average performance across all folds in the cross-validation process.

3.2 Dimensionality Reduction Algorithms

A significant challenge in applying kNN to large datasets is the computational cost associated with searching the entire training set. Additionally, noisy or irrelevant data can negatively impact the model's performance. To overcome these issues, we employ instance reduction techniques. These techniques aim to identify and select a smaller, more representative subset of the training data, leading to faster prediction times and improved accuracy [2].

A variety of rule-based techniques have been proposed in the literature to address the challenges associated with large and noisy datasets. These techniques aim to identify patterns and relationships within the data to select a subset of informative instances.

3.2.1 Condensed Nearest Neighbour Rule

Condensed nearest neighbor rules are a family of algorithms that aim to identify a minimal subset of the training data that can represent the entire dataset without significant loss of information. One prominent example is the **Generalized Condensed Nearest Neighbor** (GCNN) algorithm. GCNN iteratively selects instances that are misclassified by the current reduced set, adding them to the reduced set until no further improvement is possible. This technique effectively reduces the dataset size while preserving essential information for accurate classification.

3.2.2 Edited Nearest Neighbour Rule

Edited nearest neighbor rules, on the other hand, focus on removing noisy or outlier instances from the training data. The **Reduced Nearest Neighbor Rule with Generalized Editing** (RNGE) is a well-known example of this category. RNGE removes instances that are misclassified by their nearest neighbors. This process iteratively eliminates noisy points, leading to a cleaner and more informative dataset.

3.2.3 Hybrid Reduction Techniques

Hybrid reduction techniques combine the strengths of both condensed and edited approaches to achieve more robust and efficient reduction. The **Drop2** algorithm is a notable example of a hybrid technique. It first applies a condensed nearest neighbor rule to identify a core set of instances. Then, it uses an edited nearest neighbor rule to further refine the reduced set by removing noisy or redundant instances. This two-step process results in a compact and informative dataset.

3.3 Support Vector Machines (SVM)

This section describes the Support Vector Machines (SVM) algorithm and its implementation in our study.

3.3.1 Algorithm Overview

Support Vector Machines (SVM) is a powerful supervised learning algorithm used for classification and regression tasks. The primary objective of SVM is to find the optimal hyperplane that separates different classes in the feature space while maximizing the margin between the classes[1].

The key principles of SVM include:

- Margin Maximization: SVM aims to find the hyperplane that maximizes the margin between classes, which enhances the model's generalization capability.
- Support Vectors: The data points closest to the decision boundary, known as support vectors, play a crucial role in defining the optimal hyperplane.
- Kernel Trick: SVM can handle non-linearly separable data by mapping the input space to a higher-dimensional feature space using kernel functions.

Advantages of SVM include:

- Effectiveness in high-dimensional spaces
- Versatility through different kernel functions
- Faster consultation times than KNN, thanks to training step

Disadvantages of SVM include:

- Sensitivity to the choice of kernel function and hyperparameters
- Computational complexity for large datasets

3.3.2 Implementation Details

Unlike for KNN (see subsection 3.1 on K-Nearest Neighbors), we did not implement the SVM algorithm ourselves. Instead, we used the prebuilt implementations from the `sklearn` library [1]. Scikit-learn's `svm` module includes several different implementations of SVM:

- `SVC`: Support Vector Classification, the most commonly used implementation for classification tasks
- `NuSVC`: Support Vector Classification with Nu-SVC, similar to `SVC` but with a different formulation of the optimization problem
- `LinearSVC`: Linear Support Vector Classification, a specific variant of `SVC` that uses a linear kernel
- `SVR`: Support Vector Regression, a variant of SVM for regression tasks

For our study, we decided to use `SVC` as it is the most commonly used implementation for classification tasks, allowed for the kernel trick (unlike `LinearSVC`), and met our needs.

3.3.3 Kernel Selection

In our study, we explored multiple kernel functions to capture different types of relationships in the data. The kernels used include:

- Linear: $K(x_i, x_j) = x_i^T x_j$
- Polynomial: $K(x_i, x_j) = (\gamma x_i^T x_j + r)^d$
- Radial Basis Function (RBF): $K(x_i, x_j) = \exp(-\gamma ||x_i - x_j||^2)$
- Sigmoid: $K(x_i, x_j) = \tanh(\gamma x_i^T x_j + r)$

Where x_i and x_j are feature vectors, γ is a kernel coefficient, r is a constant term, and d is the degree of the polynomial kernel.

3.3.4 Hyperparameter Tuning

The following hyperparameters were tuned in our SVM implementation:

- C: The regularization parameter, which controls the trade-off between achieving a low training error and a low testing error. We explored values [1, 3, 5, 7].
- Kernel: We tested different kernel types including "linear", "poly", "rbf", and "sigmoid".

3.3.5 Implementation Steps

The SVM classification process in our study followed these steps:

1. Data Preparation (see section 2 on Data).
2. Model Configuration: SVM models were created with different combinations of C values and kernel types.
3. Cross-validation: For each configuration, the model was trained and evaluated using cross-validation across 10 predefined folds.
4. Performance Evaluation: Various metrics including accuracy, F1 score, and confusion matrix elements (TP, TN, FP, FN) were computed.
5. Time Measurement: Training and testing times were recorded for each configuration.
6. Results Compilation: The results for each configuration were saved in CSV files for further analysis.

[1] Scikit-learn's `svm` module documentation can be found at `https://scikit-learn.org/1.5/modules/svm.html`

Table 1: Results from KNN models for the mushroom dataset

	k	distance func	voting func	weighting func	accuracy	f1
1	7	ManhattanDistance	ShepardsWorkVote	EqualWeighting	0.951	0.952
2	5	ManhattanDistance	ShepardsWorkVote	EqualWeighting	0.951	0.952
3	3	ManhattanDistance	ShepardsWorkVote	EqualWeighting	0.951	0.952
4	1	ManhattanDistance	MajorityClassVote	EqualWeighting	0.950	0.951
5	1	ManhattanDistance	InverseDistanceWeightedVote	EqualWeighting	0.950	0.951
6	1	ManhattanDistance	ShepardsWorkVote	EqualWeighting	0.950	0.951
7	5	ManhattanDistance	InverseDistanceWeightedVote	EqualWeighting	0.933	0.933
8	7	ManhattanDistance	InverseDistanceWeightedVote	EqualWeighting	0.930	0.929
9	3	ManhattanDistance	InverseDistanceWeightedVote	EqualWeighting	0.928	0.929
10	3	ManhattanDistance	MajorityClassVote	EqualWeighting	0.926	0.927

3.3.6 Multi-class Classification

While simple SVMs are binary classifiers, they can be extended to multi-class classification through various strategies. In our case, however, both the datasets included only two classes, so we did not need to use these strategies.

3.3.7 Performance Metrics

The performance of the SVM models was evaluated using the following metrics:

- Accuracy: The proportion of correct predictions among the total number of cases examined.
- F1 Score: The harmonic mean of precision and recall, providing a balanced measure of the model's performance.
- Confusion Matrix: Including True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN).
- Training Time: The time taken to train the model.
- Testing Time: The time taken to make predictions on the test set.

The F1 score was used as our primary metric for evaluating the performance of the SVM models, since it balances the trade-off between precision and recall, and does not report overly favorable results when the dataset is imbalanced.

4 Results and Analysis

4.1 Results

Here, present the findings of the study. Include relevant data, statistics, and any figures or tables that help illustrate the results.

4.2 Discussion

In this section, interpret the results, discuss their implications, and relate them back to the research question or hypothesis. Address any limitations of the study and suggest areas for future research.

4.2.1 k-Nearest Neighbors (kNN) Analysis

This section interprets the results of the kNN algorithm, discussing its performance and implications.

Table 2: Results from KNN models for the hepatitis dataset

	k	distance func	voting func	weighting func	accuracy	f1
1	1	EuclideanDistance	ShepardsWorkVote	ReliefFWeighting	0.955	0.972
2	1	EuclideanDistance	InverseDistanceWeightedVote	ReliefFWeighting	0.955	0.972
3	1	EuclideanDistance	MajorityClassVote	ReliefFWeighting	0.955	0.972
4	1	ChebyshevDistance	ShepardsWorkVote	EqualWeighting	0.948	0.969
5	1	ChebyshevDistance	InverseDistanceWeightedVote	EqualWeighting	0.948	0.969
6	1	ChebyshevDistance	MajorityClassVote	EqualWeighting	0.948	0.969
7	7	EuclideanDistance	ShepardsWorkVote	EqualWeighting	0.948	0.968
8	5	EuclideanDistance	ShepardsWorkVote	EqualWeighting	0.948	0.968
9	1	ManhattanDistance	MajorityClassVote	EqualWeighting	0.948	0.967
10	7	ManhattanDistance	ShepardsWorkVote	EqualWeighting	0.948	0.967

Table 3: Results from SVM models for the mushroom dataset

	C	kernel type	accuracy	f1
1	1	poly	0.928	0.928
2	7	rbf	0.926	0.926
3	5	rbf	0.925	0.925
4	3	rbf	0.924	0.924
5	7	poly	0.920	0.920
6	5	poly	0.916	0.916
7	3	poly	0.916	0.915
8	3	linear	0.914	0.912
9	1	linear	0.911	0.910
10	1	rbf	0.906	0.902

Table 4: Results from SVM models for the hepatitis dataset

	C	kernel type	accuracy	f1
1	7	rbf	0.955	0.972
2	3	rbf	0.948	0.968
3	5	rbf	0.948	0.968
4	1	poly	0.948	0.968
5	3	poly	0.948	0.967
6	5	poly	0.948	0.967
7	7	poly	0.948	0.967
8	1	rbf	0.903	0.941
9	3	linear	0.890	0.929
10	1	linear	0.884	0.927

Table 5: Results from KNN models for the mushroom dataset with dimensionality reduction

	k	reduction func	accuracy	f1	train time	test time	storage
1	1	control	0.950	0.951	0.000	0.255	1000
2	1	enn	0.950	0.951	0.000	0.255	1000
3	1	drop2	0.950	0.951	0.000	0.257	1000
4	1	cnn	0.925	0.929	0.000	0.065	189

Table 6: Results from KNN models for the hepatitis dataset with dimensionality reduction

	k	reduction func	accuracy	f1	train time	test time	storage
1	1	control	0.948	0.967	0.001	0.070	1395
2	1	enn	0.948	0.967	0.000	0.072	1395
3	1	drop2	0.948	0.967	0.000	0.073	1395
4	1	cnn	0.877	0.921	0.000	0.021	409

4.2.2 Dimensionality Reduction Analysis

This section interprets the results of the dimensionality reduction techniques, discussing their impact on the analysis and visualization.

4.2.3 Support Vector Machines (SVM) Analysis

As seen in Table 3 and Table 4, SVMs performed well on both the hepatitis and the mushroom datasets, achieving peak accuracy and F1 scores which match the best results from KNN.

A key advantage of the SVM over KNN is that SVMs are much faster during consultation time

For both the hepatitis and mushroom datasets, the configuration using RBF kernel and $C = 7$ achieved outstanding accuracy and F1 scores. However, because of the simple predictability of the mushroom data, the simple Polynomial kernel performed just as well when used with $C = 1$.

5 Conclusion

Summarize the main findings of the study and their significance. Restate the key points and provide a final perspective on the research.

References

[1] Christopher J.C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2:121–167, 1998.

[2] D. Randall Wilson and Tony R. Martinez. Reduction techniques for instance-based learning algorithms. *Machine Learning*, 38(3):257–286, 2000.