Group 127

540303144, 520325186, 530419471, 550251668

September 2025

1 Introduction

1.1 Task and Dataset

This task aims to classify rice grain varieties through using multiple machine learning models. The dataset used in this task is a modified version of the UCI Rice (Cammeo and Osmancik) dataset, containing 1,400 examples described by seven morphological features extracted from grain images. Through data preprocessing, model training, and evaluation, two rice varieties—Cammeo and Osmancik will be distinguished in this task.

1.2 Classifiers Overview

The classifiers applied on the dataset used are Logistic Regression, Naïve Bayes, Decision Tree, Bagging, AdaBoost, and Gradient Boosting, K-Nearest-Neighbor (KNN) and Random Forest. Their performance is evaluated using stratified 10-fold cross-validation. Additionally, grid search is used to fine-tuning K-Nearest Neighbor (KNN) and Random Forest. This enables a systematic comparison between base models and ensemble methods.

1.3 Pipeline Summary

The overall pipeline consists of three stages: data preprocessing (imputing missing values, normalization, and encoded classes), applying multiple classification algorithms, and evaluating their predictive performance through accuracy and F1 metrics.

1.4 Key Findings

The experiment results indicate that ensemble methods, particularly Bagging and Random Forest, consistently outperform single classifiers in terms of accuracy and robustness. Logistic Regression and Naïve Bayes also achieve competitive performance, while Decision Tree is more prone to variance. These findings highlight the benefits of ensemble learning and parameter tuning in improving classification reliability on the rice dataset used in the task.

2 Method

2.1 Data Pre-Processing Procedure

The missing value in the raw dataset is denoted by "?". These were first replaced with NaN and imputed using the mean value of the column through the SimpleImputer with a mean strategy. This ensured that no samples were discarded due to incomplete information. To maintain numerical consistency between different features, all input attributes were converted to numeric types. The feature values were then normalized using Min-Max scaling, rescaling each attribute into the range [0,1]. This step prevents attributes with larger magnitudes from dominating distance-based algorithms such as KNN and improves convergence for models like Logistic Regression. Finally, the class labels, originally represented as categorical values (class1 and class2), were assigned to binary integers (0 and 1). This transformation standardizes the target variable for compatibility with scikit-learn classifiers.

2.2 Classifier

2.2.1 Logistic Regression

Intuitive Explanation. Logistic Regression is a supervised learning algorithm that is mainly used for binary classification problems, although it can be extended to multiclass settings. Unlike linear regression, which predicts a continuous output, logistic regression predicts the probability that a given input belongs to a particular class. This is achieved by applying the logistic (sigmoid) function to a linear combination of the input features, mapping values to the range $(0, 1)$, which can then be interpreted as class probabilities.

Formal Definition. The logistic regression model predicts the probability that an instance $x \in \mathbb{R}^d$ belongs to the positive class ($y = 1$) as follows:

$$P(y = 1 \mid x) = \sigma(w^T x + b) = \frac{1}{1 + e^{-(w^T x + b)}},$$

where $\sigma(z)$ is the sigmoid function, w is the weight vector, and b is the bias term. The loss function commonly used is the log-loss (cross-entropy loss):

$$L(w, b) = -\frac{1}{N} \sum_{i=1}^{N} \Big[y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i) \Big],$$

where $\hat{y}_i = P(y = 1 \mid x_i)$ is the predicted probability for sample i.

Technical Details. Logistic regression is a linear model in the feature space, which makes it simple and efficient. It assumes a linear decision boundary between classes, so it may under perform on highly nonlinear datasets. Regularization terms such as L_1 (Lasso) or L_2 (Ridge) are often added to the loss

function to avoid overfitting and improve generalization. The model parameters are usually estimated using maximum likelihood estimation, optimized through algorithms such as gradient descent or variants like stochastic gradient descent (SGD).

Implementation. We employed the `logregClassifier` implementation from scikit-learn (version 1.4). The model performance was evaluated with stratified 10-fold cross-validation to preserve class distribution across folds. The evaluation metric was classification accuracy. The Logistic Regression classifier achieved an average cross-validation accuracy of **93.86%**.

2.2.2 Naive Bayes.

Naïve Bayes is a family of probabilistic classifiers based on Bayes' theorem with a strong assumption of conditional independence between features. Despite its simplicity, it often performs surprisingly well in practice, especially for high-dimensional datasets such as text classification. The algorithm estimates the posterior probability of a class given the input features and assigns the class with the highest probability.

Formal Definition. Given a data sample $x = (x_1, x_2, \ldots, x_d)$, the posterior probability of class C_k is computed using Bayes' theorem:

$$P(C_k \mid x) = \frac{P(C_k) \prod_{i=1}^{d} P(x_i \mid C_k)}{P(x)}.$$

The denominator $P(x)$ is constant for all classes and thus ignored during prediction. The classification rule is:

$$\hat{y} = \arg\max_{C_k} P(C_k) \prod_{i=1}^{d} P(x_i \mid C_k).$$

In the Gaussian Naïve Bayes variant, each conditional probability $P(x_i \mid C_k)$ is assumed to follow a Gaussian distribution with mean μ_{ik} and variance σ_{ik}^2:

$$P(x_i \mid C_k) = \frac{1}{\sqrt{2\pi\sigma_{ik}^2}} \exp\left(-\frac{(x_i - \mu_{ik})^2}{2\sigma_{ik}^2}\right).$$

Technical Details. Naïve Bayes is computationally efficient, with training complexity that scales linearly with the number of features. It works well for large and high-dimensional datasets, since it only requires estimating simple class-conditional probabilities. The independence assumption simplifies probability estimation but may reduce accuracy when features are strongly correlated. Different variants exist for handling different data types: Gaussian Naïve Bayes for continuous attributes, Multinomial Naïve Bayes for count data, and Bernoulli Naïve Bayes for binary features. Although the independence assumption is rarely satisfied in real-world datasets, the algorithm often performs competitively and serves as a strong baseline for classification tasks.

Implementation. In this study, the Gaussian Naïve Bayes classifier from the `scikit-learn` library was applied. Model evaluation was carried out using stratified 10-fold cross-validation, with classification accuracy as the evaluation metric. The Naïve Bayes classifier achieved an average cross-validation accuracy of **92.64%**

2.2.3 Decision Tree

Intuitive Explanation. Decision Tree is a non-parametric supervised learning algorithm which is used for classification and regression. A decision tree break down the feature space step by step into smaller areas by posing a series of questions(splits). Locally at each node, the optimal feature and threshold value that best divide the data are selected, and the process continues until the tree structure is obtained where the leaves correspond to class labels.

Formal Definition. Splitting criterion is derived through information gain, which is one of the measure that means decrease in entropy:

$$IG(S, A) = H(S) - \sum_{v \in \text{ Values } (A)} \frac{|S_v|}{|S|} H\left(S_v\right),$$

where $H(S) = -\sum_c p_c \log p_c$ is the entropy of dataset S, and S_v is the subset of S for which attribute A has value v.

Technical Details. Decision trees are prone to overfitting if grown without constraints. Limiting the maximum depth reduces model complexity but may increase bias. Using entropy as the splitting criterion ensures splits are chosen to maximize information gain, which is equivalent to minimizing class impurity. While decision trees are interpretable and flexible, they are sensitive to small changes in data, which can lead to instability in predictions.

Implementation. We employed the `DecisionTreeClassifier` implementation from scikit-learn (version 1.4). The model was instantiated with `criterion="entropy"` and `random_state=0` to ensure reproducibility. Model evaluation was conducted using stratified 10-fold cross-validation, and the mean accuracy across folds was reported as the performance measure.

2.2.4 Bagging

Intuitive Explanation. Bagging is an ensemble method that reduces variance by training multiple base classifiers on bootstrap samples and combining their predictions.Each base learner is trained on a random sample (with replacement) drawn from the training set. Because different learners see different subsets of the data, the ensemble prediction—obtained by majority voting—becomes more stable than any single tree.

Formal Definition. Given M base classifiers $h_1(x), h_2(x), \ldots, h_M(x)$ trained on bootstrap samples, the Bagging ensemble predicts as:

$$H(x) = \text{majority\_vote}\big(h_1(x),\, h_2(x),\, \ldots,\, h_M(x)\big).$$

Technical Details. Bagging reduces variance by aggregating predictions from diverse base learners. Decision trees are a natural choice as base learners due to their high variance and sensitivity to training data. The number of trees (`n_estimators`), the sample size per learner (`max_samples`), and the maximum tree depth (`max_depth`) govern the trade-off between bias and variance. Larger ensembles typically yield more stable predictions, but at higher computational cost.

Implementation. We employed the `BaggingClassifier` implementation from scikit-learn (version 1.4). The base learner was a decision tree instantiated with `criterion="entropy"`, `max_depth=5`, and `random_state=0`. The Bagging ensemble was configured with `n_estimators=50`, `max_samples=100`, and `bootstrap=True`. Model performance was evaluated using stratified 10-fold cross-validation, and the mean accuracy across folds was reported.

2.2.5 AdaBoost

Intuitive Explanation. AdaBoost (Adaptive Boosting) is an ensemble method that combines many weak classifiers into a single strong classifier. Each weak classifier is only slightly better than random guessing, but AdaBoost sequentially focuses more on the difficult samples that previous classifiers misclassified. By assigning higher weights to these hard samples and giving more influence to accurate classifiers, the algorithm produces a final model that achieves high accuracy through a weighted majority vote.

Formal Definition. Given a binary classification dataset

$$\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}, \quad x_i \in \mathbb{R}^d,\ y_i \in \{-1, +1\},$$

AdaBoost proceeds as follows.

Step 1. Initialization. Assign equal weights:

$$w_i^{(1)} = \tfrac{1}{N}, \quad i = 1, \ldots, N.$$

Step 2. Iteration. For $t = 1, \ldots, T$:

1. Train weak classifier h_t with weighted data.
2. Compute weighted error:

$$\epsilon_t = \sum_{i=1}^{N} w_i^{(t)}\, \mathbf{1}\{y_i \neq h_t(x_i)\}.$$

3. Set classifier weight:

$$\alpha_t = \tfrac{1}{2} \ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right).$$

4. Update sample weights:

$$w_i^{(t+1)} = \frac{w_i^{(t)} \exp(-\alpha_t y_i h_t(x_i))}{Z_t},$$

with Z_t normalizing to $\sum_i w_i^{(t+1)} = 1$.

Step 3. Final Classifier. The strong classifier is the weighted majority:

$$H(x) = \operatorname{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right).$$

Technical Details. AdaBoost was implemented with decision stumps (`max_depth = 1`) as weak learners. The number of boosting rounds T and the learning rate η were fixed in advance. These settings ensure that each base learner contributes only marginally, consistent with the theoretical design of AdaBoost. The `SMME` framework was adopted rather than the standalone `SMME.R` script, as it provides a more stable and standardised environment for model training and evaluation.

Implementation. We used the `AdaBoostClassifier` from `scikit-learn` (v1.4). Decision stumps were employed as weak learners, and the specified boosting parameters were fixed a priori. Model performance was assessed solely by the mean accuracy obtained under stratified 10-fold cross-validation. The choice of `SMME` ensured reproducibility and compatibility across models, avoiding the additional maintenance overhead associated with a separate `SMME.R` script.

2.2.6 Gradient Boosting

Intuitive Explanation. Gradient Boosting is a boosting-based ensemble method that builds an additive model in a forward stage-wise manner. Instead of reweighting samples like AdaBoost, Gradient Boosting trains each new weak learner to predict the residual errors (negative gradients) of the current model with respect to a specified loss function. By iteratively fitting new learners to these residuals, the ensemble progressively reduces overall prediction error.

Formal Definition. Given a differentiable loss function $L(y, F(x))$, Gradient Boosting builds the additive model

$$F_M(x) = \sum_{m=1}^{M} \gamma_m h_m(x),$$

where each $h_m(x)$ is a decision tree. At stage m, pseudo-residuals are computed as the negative gradients:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F=F_{m-1}},$$

and a new tree $h_m(x)$ is trained to fit these residuals. The optimal step size γ_m is determined by line search:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\big(y_i,\, F_{m-1}(x_i) + \gamma h_m(x_i)\big).$$

Technical Details. Gradient Boosting reduces bias and improves accuracy by sequentially correcting residual errors. The learning rate η scales the contribution of each weak learner, with smaller values typically requiring more iterations but improving generalization. The number of boosting stages (`n_estimators`) controls model complexity. While powerful, Gradient Boosting can be computationally expensive and sensitive to overfitting if hyperparameters are not carefully tuned.

Implementation. We employed the `GradientBoostingClassifier` from scikit-learn (version 1.4). The model was configured with `n_estimators=50`, `learning_rate=0.5`, and `random_state=0`. For binary classification, the default logistic loss was used. Model performance was assessed using stratified 10-fold cross-validation, and the mean accuracy across folds was reported.

2.2.7 K-Nearest Neighbor

Intuitive Explanation. The *K-Nearest Neighbor (KNN)* algorithm is a simple instance-based learning method that classifies a sample based on the majority label among its k nearest training examples in the feature space. Distance is typically measured using either the Manhattan ($p = 1$) or Euclidean ($p = 2$) metric. Intuitively, the method assumes that samples with similar feature values are likely to belong to the same class.

Formal Definition. Formally, given a query instance x, the algorithm computes the distance $d(x, x_i)$ between x and every training instance x_i. It then selects the set $N_k(x)$ of the k closest training samples. The predicted class $\hat{y}$ is determined by majority voting:

$$\hat{y} = \arg\max_{c \in \{0,1\}} \sum_{x_i \in N_k(x)} \mathbb{I}(y_i = c),$$

where $\mathbb{I}(\cdot)$ is the indicator function and y_i is the label of training instance x_i. The distance function is defined by the Minkowski metric:

$$d(x, x_i) = \left(\sum_{j=1}^{m} |x_j - x_{i,j}|^p \right)^{\frac{1}{p}},$$

with $p = 1$ corresponding to Manhattan distance and $p = 2$ corresponding to Euclidean distance.

Technical Details. K-Nearest Neighbor (KNN) is a non-parametric, instance-based learning algorithm that classifies a query sample based on the majority vote of its k nearest neighbors in the feature space. The distance metric is crucial: with $p = 1$, the Manhattan distance is used, while $p = 2$ corresponds to the Euclidean distance. Smaller values of k allow the model to capture local patterns but may increase sensitivity to noise, whereas larger k smooths decision boundaries at the cost of finer granularity. Since KNN has no explicit training phase, computational cost arises mainly during prediction, making scalability dependent on dataset size.

Implementation. We implemented the `KNeighborsClassifier` from `scikit-learn` (version 1.5). The grid search explored $k \in \{1, 3, 5, 7\}$ and $p \in \{1, 2\}$ under stratified 10-fold cross-validation. The model was trained and evaluated with `random_state=0` to ensure reproducibility. Model performance was reported in terms of cross-validation accuracy, along with test set accuracy for the optimal parameter pattern.

2.2.8 Random Forest

Intuitive Explanation. The *Random Forest (RF)* algorithm is an ensemble learning method that combines multiple decision trees to improve predictive accuracy and reduce overfitting. Each tree is trained on a bootstrap sample of the training data, and at each split a random subset of features is considered, which introduces diversity among the trees. The final prediction is obtained by majority voting across all trees. This approach reduces the variance of individual decision trees while maintaining low bias. In this assignment, the implementation was carried out using the `RandomForestClassifier` from the `scikit-learn` library, with information gain (entropy) as the splitting criterion and `max_features` set to `"sqrt"`.

Formal Definition. Formally, let $\{T_b(x)\}_{b=1}^{B}$ denote the set of B decision trees constructed on bootstrap samples of the training data. Each tree T_b is grown by recursively splitting nodes using the entropy criterion:

$$IG(S, A) = H(S) - \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} H(S_v),$$

where $IG(S, A)$ is the information gain of splitting dataset S on attribute A, and $H(S)$ is the entropy:

$$H(S) = -\sum_{c \in \{0,1\}} p_c \log p_c,$$

with p_c the proportion of class c in S.

The Random Forest prediction for a query sample x is obtained by aggregating the predictions of all trees via majority voting:

$$\hat{y} = \arg \max_{c \in \{0,1\}} \sum_{b=1}^{B} \mathbb{I}(T_b(x) = c),$$

where $\mathbb{I}(\cdot)$ is the indicator function. By combining multiple diverse trees, the Random Forest achieves higher stability and accuracy compared to a single decision tree.

Technical Details. Random Forest (RF) is an ensemble method that constructs a collection of decision trees trained on bootstrap samples of the dataset. At each split, a random subset of features is considered, introducing additional randomness that reduces correlation among trees. The final prediction is obtained by majority voting across all trees. The number of trees ($n\_estimators$) determines ensemble size and stability, while the maximum number of leaf nodes (max_leaf_nodes) controls tree depth and model complexity. This design improves generalisation by reducing variance relative to a single decision tree, while retaining interpretability and robustness to noise.

Implementation. We employed the `RandomForestClassifier` from `scikit-learn` (version 1.5), using the entropy criterion for information gain and `max_features="sqrt"`. A parameter grid was defined with $n\_estimators \in \{10, 30, 60, 100\}$ and $max\_leaf\_nodes \in \{6, 12\}$. Hyperparameter optimisation was conducted with `GridSearchCV` under stratified 10-fold cross-validation. After tuning, the best model was retrained on the training set and evaluated on the independent test set, with performance reported using accuracy, macro F1, and weighted F1 scores.

2.3 Evaluation Procedure

Overview To ensure a fair and consistent comparison across models, all classifiers were evaluated using stratified 10-fold cross-validation. The dataset was partitioned into ten equally sized folds while preserving the class distribution in each fold. For each iteration, nine folds were used for training and the remaining fold for validation, with the process repeated ten times so that each fold served as validation exactly once. The final cross-validation accuracy was reported as the mean of the ten validation scores.

Part 1 In the first stage, six baseline classifiers were implemented with their default hyperparameter settings: Logistic Regression, Naïve Bayes, Decision Tree, Bagging, AdaBoost, and Gradient Boosting. No hyperparameter optimisation was performed at this stage. The sole evaluation criterion was the mean stratified 10-fold cross-validation accuracy, which provides a direct measure of predictive performance under standard configurations.

This baseline comparison establishes a reference point against which the benefits of hyperparameter tuning and additional evaluation metrics in Part 2 can be assessed.

Part 2 For *K-Nearest Neighbour (KNN)* and *Random Forest (RF)*, hyperparameter optimisation was performed using `GridSearchCV`, also under stratified 10-fold cross-validation. The grid search exhaustively explored parameter combinations (e.g., k and p for KNN, $n\_estimators$ and max_leaf_nodes for RF), selecting the configuration that maximised the mean cross-validation accuracy.

After hyperparameter tuning, the dataset was split into training and test subsets using stratified sampling with `random_state=0`. The optimised models were retrained on the training subset and evaluated on the independent test subset to assess generalisation performance.

Performance metrics included:

- **Accuracy**: the proportion of correctly classified samples.
- **Macro and Weighted F1-scores** (for Random Forest): to account for class imbalance and provide complementary perspectives on precision and recall.

This procedure ensures reproducibility, controls randomness, and provides both cross-validation and independent test evaluations for robust performance analysis.

3 Experiments

3.1 Experimental Setup

All experiments were conducted using **Python 3.11** in a **Jupyter Notebook** environment. The implementation relied on the `scikit-learn` (v1.5), `numpy` (v1.26), and `pandas` (v2.2) libraries, ensuring reproducibility and compatibility with standard machine learning workflows.

The dataset used was `rice-final2.csv`, a modified version of the UCI Rice dataset, containing **1,400 samples** with **7 numerical features** extracted from rice grain images. Each sample belonged to one of two classes, representing the Cammeo and Osmancik rice varieties.

Prior to training, the dataset underwent preprocessing:

- Missing values were imputed with the mean of each column using `SimpleImputer`.

- Features were normalised into the range $[0, 1]$ using `MinMaxScaler`.
- Class labels were mapped from categorical values (`class1`, `class2`) to binary integers (0,1).

For evaluation, models were trained and tested under the following conditions:

- **Cross-validation**: stratified 10-fold cross-validation was applied to preserve class distribution across folds.
- **Train/test split**: for KNN and Random Forest, the dataset was split into training and test subsets with stratified sampling and `random_state=0`.
- **Parameter grids**:
  - KNN: $k \in \{1, 3, 5, 7\}$, $p \in \{1, 2\}$ (Manhattan and Euclidean distance).
  - Random Forest: $n\_estimators \in \{10, 30, 60, 100\}$, $max\_leaf\_nodes \in \{6, 12\}$, with `max_features="sqrt"`.

Performance was primarily measured using **classification accuracy**. For Random Forest, additional metrics included **macro-average F1** and **weighted-average F1**, to capture class-level balance in prediction performance.

3.2 Experimental Results

Model	**Average Cross-Validation Accuracy (%)**
Logistic Regression	0.9386
Naïve Bayes	0.9264
Decision Tree	0.9179
Bagging	0.9414
AdaBoost	0.9257
Gradient Boosting	0.9321

Table 1: Performance of classification models under 10-fold cross-validation without parameter tuning.

Metric	KNN	Random Forest
Best hyperparameters	$k = 5$ $p = 1$	$n\_estimators = 10$ $max\_leaf\_nodes = 6$
Test macro F1	—	0.9293
Test weighted F1	—	0.9308
Cross-validation accuracy	0.9357	0.9418
Test accuracy	0.9333	0.9310

Table 2: Comparison of KNN and Random Forest with hyperparameter tuning under 10-fold cross-validation.

3.3 Discussion of Results

According to the table, Logistic Regression and Naïve Bayes achieved strong performance (93.86% and 92.64% respectively), which suggests that the rice dataset is relatively well-structured and linearly separable. However, Decision Tree alone performed slightly worse (91.79%), reflecting its higher variance when used as a single learner.

Ensemble methods demonstrated improved robustness and accuracy. Bagging achieved the best cross-validation performance (94.14%), indicating that aggregating multiple decision trees effectively reduced variance. Gradient Boosting followed with 93.21%, balancing bias reduction and predictive power. AdaBoost achieved 92.57%, showing benefits from sequential reweighting but slightly underperforming compared to Bagging.

For the results of hyperparameter optimisation, KNN with $k = 5$, $p = 1$ achieved 93.57% cross-validation accuracy and 93.33% test accuracy, confirming its competitiveness when parameters are carefully tuned. Random Forest achieved the highest cross-validation accuracy at 94.18% with $n_estimators = 10$ and $max_leaf_nodes = 6$, although its test accuracy (93.10%) was slightly lower than KNN. Importantly, Random Forest achieved strong macro and weighted F1 scores (0.9293 and 0.9308), showing consistent classification across both classes.

4 Conclusion

4.1 Summary of Findings

This task evaluated various classifers on the rice dataset including single, ensembled and fine-tunning models. The results showed that ensemble methods consistently outperformed single classifiers. Bagging achieved the best overall cross-validation accuracy (94.14%), while Random Forest achieved the highest tuned cross-validation accuracy (94.18%) with balanced F1 scores. Logistic Regression and Naïve Bayes also demonstrated strong performance (93.86% and 92.64%), suggesting the dataset's separability. KNN performed competitively

when tuned ($k = 5$, $p = 1$), achieving 93.33% test accuracy. Overall, ensemble methods, particularly Bagging and Random Forest, proved to be the most robust and reliable approaches.

4.2 Limitations

In this task, the dataset contains only 1,400 samples across two rice varieties, which may limit the generalisability of the findings. Moreover, only a subset of classification algorithms and parameter ranges were explored, meaning potentially better configurations may have been ignored.

4.3 Future Work

Future work could address these limitations by using the original rice dataset to include larger sample sizes. Exploring additional algorithms such as Neural Networks may provide further performance improvements. Broader hyperparameter tuning ranges, for example number of k in K-NN model.

References

[1] Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., ... & Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, 585(7825), 357–362.

[2] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12, 2825–2830.

[3] Wes McKinney. (2010). Data structures for statistical computing in Python. *Proceedings of the 9th Python in Science Conference*, 51–56.