Conformal Prediction of Multi-Class Structured Label Spaces

Teja Reddy Kasireddy

Submitted for the Degree of Master of Science in

Artificial Intelligence

Department of Computer Science
Royal Holloway University of London
Egham, Surrey TW20 0EX, UK

Aug 30, 2023

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 11484

Student Name: Teja Reddy Kasireddy

Date of Submission: 30/08/23

Signature: Teja Reddy

Abstract

This research paper presents an in-depth investigation into the application of conformal prediction in the context of multi-class classification using machine learning models. The study revolves around the classification of images of Greek letters as well as the CIFAR-100 dataset. The primary focus is to explore the efficacy of various conformity measures in enhancing the reliability and accuracy of predictions.

The research begins by selecting appropriate datasets for experimentation and establishing the label space structure. A detailed exploration of the datasets reveals their characteristics, balanced class distribution, and feature representation. The Greek character dataset is used for the majority of the analysis, while the CIFAR-100 dataset serves as an alternative benchmark for the final phase. Multiple machine learning models, including Support Vector Machines (SVM), Random Forests, Logistic Regression, and Bagging classifiers, are implemented to establish baseline performances. Hyperparameter tuning is applied to optimize each model's performance. Evaluation metrics such as accuracy, confusion matrices, and classification reports are used to assess their effectiveness.

The paper introduces the concept of conformity scores, an essential component of conformal prediction. The conformity measures examined include basic conformity, semantic-based conformity, and inductive conformity measures. The incorporation of semantic information, despite limited data availability, is explored through the identity matrix. Five specialized inductive conformity measures are introduced: Same Class, Weighted Distances, Distances Product, TR-Multiplication, and TR-Exponent. The research delves into the generation of conformal prediction sets, which provide a range of plausible labels for each test sample. These prediction sets are based on the calculated conformity scores and aim to enhance the reliability of predictions, especially in cases of low model accuracy. The interpretation and analysis of the results provide insights into the strengths and limitations of each conformity measure.

The conclusions drawn from the study shed light on the challenges posed by low-accuracy models. The analysis of prediction sets reveals the varying behavior of different conformity measures and their influence on prediction reliability. While some measures produce conservative prediction sets, others are more expansive, reflecting the models' uncertainty.

Acknowledgement

I would like to express my heartfelt gratitude to all those who have supported me throughout my journey. Their unwavering encouragement and guidance have played a pivotal role in my accomplishments.

I am deeply thankful to my supervisor, Ilia Nouretdinov, whose mentorship and insights have been invaluable. Your expertise and dedication have shaped my growth in profound ways.

To my parents, K. Venkatasubbareddy and K. Ramasubbama, I owe an immeasurable debt of gratitude. Your constant love, belief, and sacrifices have been the foundation of my success. You've shown me the importance of determination and hard work.

My siblings, K. Giri Naga Prasad Reddy and K. Balaji, have always been a source of motivation and camaraderie. Your support has given me the strength to persevere, even in challenging times.

Contents

1 Introduction

1.1 Overview

In an era where data is becoming increasingly complex, reliable machine learning techniques for multi-class classification in structured label spaces are more critical than ever. Applications span diverse fields from medical diagnostics to material science, often requiring not just accurate but also interpretable and reliable predictions. This project aims to bridge this gap by employing conformal prediction methods, tailored to account for structured label spaces. The end goal is to select a suitable dataset, such as Greek character recognition or CIFAR-100, and implement a robust machine learning algorithm that leverages special conformity score functions. This work promises to make a substantive contribution to the technical community by providing a more reliable and nuanced framework for multi-class classification, enhancing the system's applicability and trustworthiness across various sectors.

1.2 Need for Validation of ML models

Conformal predictors offer a way to gauge the reliability of machine learning predictions by providing not just point estimates but also prediction regions with confidence levels. They're particularly useful in situations of high uncertainty or limited training data, where traditional models might falter.

Example: In a medical diagnosis system, a traditional model might say a patient is diagnosed with a condition. But what if the patient's features are borderline? Conformal predictors help by providing a prediction interval, not just a binary answer. So instead of just "diagnosed," you'd get a range of possibilities with associated confidence levels, aiding medical professionals in making more nuanced decisions. The technique uses the distribution and reliability information from the training data to compute these prediction intervals. This makes conformal predictors valuable in areas like medical diagnosis and financial risk, where both accuracy and confidence are paramount.

1.3 Aim and Objectives

To select and critically analyze a dataset suitable for classification tasks featuring a large number of classes (ranging from 20 to 200). (CIFAR100 and Greek). Evaluate available datasets such as Greek character recognition and CIFAR-100 to identify one that is most relevant to the project's aim.

Choose a basic machine learning algorithm that can be adapted to multi-class classification problems. Extend the chosen algorithm to accommodate multi-class labels and obtain preliminary results. Prepare an in-depth analysis of the selected dataset and machine learning techniques relevant to structured label space.

Create conformity score functions that consider the structure in the label space, such as semantic similarity metrics. Implement a conformal prediction algorithm optimized for the chosen dataset. Assess the algorithm's output, focusing on its reliability and effectiveness in dealing with structured label spaces.

2 Literature Review

2.1 Machine Learning

Machine learning is a subset of artificial intelligence that provides systems the ability to automatically learn and improve from experience without being explicitly programmed. This learning process is based on the recognition of complex patterns in data and the making of intelligent decisions based on them. While the landscape of machine learning is vast and continually evolving, some fundamental algorithms serve as the building blocks for a variety of complex applications. These include Decision Trees, which offer simplicity and interpretability in tackling both classification and regression tasks. Support Vector Machines (SVM) extend the capabilities to higher dimensions and are widely used for classification tasks, offering robustness against overfitting. K-Nearest Neighbours (K-NN) provides a straightforward, yet effective, method for classification and regression tasks by relying on the local structure of the data.

Advanced ensemble methods like XG-Boost and Bagging further fortify predictive models by combining multiple algorithms, thereby improving accuracy and reducing overfit. Deep learning models, such as MobileNetV2 and ResNet50, leverage neural networks with multiple layers to perform exceptionally well in tasks like image and speech recognition. Conformity measures, varying from basic to semantic-based scores, add an additional layer of evaluation, providing insights into the model's prediction confidence and semantic understanding. Each of these algorithms and techniques has its unique advantages, disadvantages, and areas of application, offering a rich toolkit for researchers and practitioners alike to solve an extensive range of problems.

2.2 Random Forests

Random Forest (Mbaabu, 2020) is a machine learning technique that improves predictive accuracy by aggregating multiple decision trees. This ensemble approach not only enhances accuracy but also addresses the overfitting issue common in individual decision trees. Unlike other ensemble methods like bagging, Random Forest further randomizes the model by allowing each decision tree to only consider a random subset of features at each split, making it more robust.

Feature importance is a standout capability of Random Forest models. It quantifies the impact of each feature in the dataset on the predictive model. Calculated as the product of the decrease in entropy or the Gini index and the probability of reaching a particular node, this score is invaluable for feature selection. By ranking features according to their importance, it becomes easier to eliminate irrelevant or redundant features, thereby reducing the risk of overfitting.

Random Forest models have their pros and cons. On the upside, they are versatile, applicable to both classification (Mbaabu, 2020) (Ng, 2023) and regression problems, and are relatively straightforward if one is already familiar with decision trees. They are particularly good at controlling overfitting. On the downside, they can become computationally expensive when using a large number of estimators, which might limit their real-world applications. Additionally, while they excel in predictive power, they lack the ability to interpret relationships between variables.

2.3 Support Vector Machines

Support Vector Machines (SVM) (Ng, 2023) are a class of supervised learning algorithms used for classification and regression tasks. The core idea is to find a hyperplane that best separates different classes in the feature space. SVM is particularly useful in high-dimensional spaces and is capable of handling non-linear relationships through the use of different kernel functions, like polynomial or radial basis function (RBF) kernels. It performs well when there's a clear margin of separation between classes, making it a popular choice in cases where precision is a priority.

SVM models also allow for nuanced control over decision boundaries through the use of hyperparameters, such as the regularization term (C), and kernel parameters. Tuning these hyperparameters can significantly impact the model's performance. For example, a high value of C will fit the training data as closely as possible, while a low value will create a larger margin, potentially allowing some misclassification for the sake of generalization. These hyperparameters can be optimized through techniques like grid search or cross-validation to produce a model best suited for the specific problem at hand.

Despite its strengths, SVM does come with limitations. It can be computationally expensive, particularly with large datasets and high-dimensional feature spaces. This may restrict its applicability in real-time or large-scale applications. Additionally, while SVM models can be incredibly accurate, they often suffer from a lack of interpretability; the decision boundaries, especially in higher dimensions or when using complex kernels, can be difficult to understand or explain. Therefore, while SVMs are powerful tools for predictive modelling, they may not be the best choice for applications requiring transparency or real-time predictions.

2.4 K- Nearest Neighbour

K-Nearest Neighbours (K-NN) (Scaler, 2023) is a type of instance-based learning algorithm used mainly for classification, though it can also be applied to regression tasks. The algorithm operates on the principle that similar data points in the feature space are likely to have the same class label. For a given data point, K-NN identifies the 'k' nearest data points in the training set and classifies the point based on a majority vote from its neighbours. The algorithm is non-parametric, meaning it makes no explicit assumptions about the underlying data distribution, and it's also highly interpretable due to its simple and intuitive nature.

One of the key advantages of K-NN is its ease of use and quick implementation. Since it doesn't require any training phase, it can be immediately applied to new data points. However, this simplicity comes at a cost. The choice of 'k' and the distance metric (Euclidean, Manhattan, etc.) can profoundly affect the model's performance. Smaller values of 'k' will capture noise and lead to overfitting, while larger values may dilute the class boundaries. To find the optimal 'k', cross-validation methods are commonly employed.

Despite its ease and adaptability, K-NN has several drawbacks. Firstly, it can be computationally intensive, particularly for large datasets, as it requires the calculation of distances to every point in the dataset for each query point. Secondly, K-NN doesn't inherently handle imbalanced class distributions well, which can skew the majority vote. Lastly, the algorithm is sensitive to irrelevant or correlated features, which can distort the distances and lead to incorrect classifications. Therefore, while K-NN may be a good starting point for classification tasks and quick prototyping, its limitations should be carefully considered in more complex or large-scale applications.

2.5 XG-Boost

Extreme Gradient Boosting (XG Boost) (nvidia, 2023) is an optimized implementation of gradient-boosted trees, designed for speed and performance. It's an ensemble learning method that builds upon the concept of boosting, wherein weak learners (typically decision trees) are combined to create a strong predictive model. XG Boost is particularly known for its scalability and ability to run on distributed systems. It excels in various types of data science problems, including classification, regression, and ranking tasks. Unlike simpler algorithms, it has built-in mechanisms for handling missing data and allows for the use of custom optimization objectives and evaluation criteria.

One of the key strengths of XG Boost is its flexibility and feature-rich nature. The algorithm offers various hyperparameters like learning rate, max depth of trees, and regularization terms, providing extensive control over the model's performance. Tuning these hyperparameters correctly can lead to highly accurate models. It also includes functionalities for feature importance, helping in feature selection and understanding the impact of different features on the target variable. Built-in techniques like column block and regularization help prevent overfitting, making the model robust.

However, XG Boost is not without its limitations. The algorithm can be computationally intensive, particularly when tuning a large number of hyperparameters or dealing with very large datasets. This can make it less suitable for real-time predictions or applications with computational constraints. Additionally, the complexity and multitude of hyperparameters can make the model hard to interpret, especially for those not intimately familiar with the algorithm's workings. Thus, while XG Boost is a powerful tool for predictive modelling, it demands careful tuning and a thorough understanding of its hyperparameters for optimal performance.

2.6 Bagging

Bagging, or Bootstrap Aggregating (Rai, 2023), is an ensemble learning technique aimed at improving the stability and accuracy of machine learning algorithms. In the training phase, bagging utilizes bootstrapped samples of the original dataset to train a series of base models, commonly decision trees. These base models are then aggregated during the prediction phase, producing a single fitted value that is either a mean value for regression problems or a majority-voted value for classification tasks. The key idea is to train each model on a different sample and then average out their predictions, thereby reducing the model variance and potentially increasing prediction accuracy.

One of the main advantages of the bagging approach is its capacity to combine multiple weak learners, resulting in a model that often outperforms single strong learners. It is particularly effective in reducing overfitting, a common problem in machine learning, and can be applied to both classification and regression problems. Moreover, because bagging uses bootstrapped samples of the original data, it offers a way to estimate values like the mean more robustly when the available dataset is limited. However, this strength comes with trade-offs. Bagging models can be computationally expensive, especially with a large number of base estimators, which can make them less practical for real-time predictions or other resource-constrained applications.

While bagging models (Rai, 2023) are often highly accurate, they do have their shortcomings. For one, they sacrifice model interpretability, making it challenging to decipher how decisions are made or which features are most impactful. Also, although bagging typically reduces variance, it can introduce bias into the model, potentially leading to underfitting. This trade-off between bias and variance is an inherent feature of bagging and should be carefully considered during model selection and tuning. Overall, bagging is a powerful technique but requires careful implementation and consideration of its limitations.

2.7 MobileNetV2

MobileNetV2 (Chen, n.d.) is a neural network architecture optimized for mobile and embedded vision applications, and it's available as a pre-trained model in TensorFlow's Keras library. This architecture is designed to be lightweight and efficient, with the primary aim of enabling high-performance machine learning models to run on resource-constrained devices. It builds on the original MobileNet architecture by incorporating inverted residual blocks with linear bottlenecks, which improves both computational efficiency and model accuracy. While the original MobileNet used depthwise separable convolutions to achieve a smaller model size and faster computation, MobileNetV2 further refines the structure to improve performance without significantly increasing computational burden.

One of the major advantages of MobileNetV2 is its efficiency at a low computational cost. This makes it ideal for real-time applications on mobile devices or embedded systems where computational resources are limited. It's suitable for a wide array of vision tasks, including but not limited to image classification, object detection, and semantic segmentation. The architecture is highly modular, allowing for easy customization and adaptability. Pre-trained models are available for immediate use, and these can be fine-tuned on a specific task to improve performance. However, while MobileNetV2 is efficient, it may not always reach the same level of accuracy as larger, more complex architectures when computational resources are abundant.

Despite its many strengths, MobileNetV2 does have limitations. For highly complex tasks or scenarios where computational resources are not a primary concern, other architectures like ResNet or VGG might deliver better performance. Furthermore, while the architecture itself is designed for efficiency, the actual speed and resource consumption can be dependent on the specific hardware it's running on, so performance gains are not guaranteed across all platforms. In summary, MobileNetV2 offers a strong balance of computational efficiency and model performance, but like any architecture, it has its own set of trade-offs that should be considered based on the specific requirements of a project.

2.8 ResNet50

ResNet50 (Kaiming He, 2015) is a variant of the Residual Network (ResNet) architecture and is readily available as a pre-trained model in TensorFlow's Keras library. It consists of 50 layers and is known for its deep structure, enabled by the introduction of "skip connections" or "residual connections" that allow gradients to flow through the network more effectively. These residual connections help mitigate the vanishing gradient problem, which is common in deep neural networks, thereby enabling the model to learn from the data more effectively. The architecture is well-suited for a wide range of image recognition tasks, including classification, localization, and detection.

The key advantage of ResNet50 lies in its depth and complexity, allowing the model to learn more nuanced features and achieve high accuracy in various vision tasks. It's particularly effective for complex tasks that require high computational power and memory. Pre-trained versions of the model offer a quick way to apply transfer learning, where the model's existing knowledge can be fine-tuned to adapt to specific tasks, thereby speeding up the training process. However, the computational intensity of ResNet50 can be a drawback in scenarios where resources are limited, or real-time inference is required.

Despite its excellent performance metrics, ResNet50 does have limitations. Its complexity and size make it computationally expensive, both in terms of memory and processing power. This might not make it the ideal choice for mobile or embedded applications where resources are constrained. Also, the extensive depth of the model, while aiding in learning complex features, could lead to longer training times, especially on limited hardware. In summary, ResNet50 offers high accuracy and is capable of handling complex tasks but comes with computational overhead that may not be suitable for all applications.

2.9 Conformity Scores

Conformity scores (Bates, 2022) are essential for conformal prediction. The models are as follows:

Basic Conformity Scores with Class Probability Estimates

Function: `conformity_scores_basic = 1 - np.max(y_pred, axis=1)`

Importance: This function calculates basic conformity scores, providing a quick metric to evaluate the confidence level of each prediction. It is effective for identifying high-confidence and low-confidence predictions, which can inform subsequent model adjustments or decision-making processes.

Relevance: In a research context, this function serves as a baseline metric for model evaluation. Its simplicity makes it a useful point of comparison when introducing more complex conformity measures.

Identity Matrix for Semantic-based Conformity Scores

Function: `semantic_matrix = np.identity(100)`

Importance: The function creates an identity matrix, which is a rudimentary way to incorporate semantic information into the conformity score calculations. It acts as a placeholder when richer semantic information is not available.

Relevance: In the scope of research, this could be the starting point for experimenting with more nuanced semantic matrices that account for hierarchical or ontological relationships between classes.

Semantic-based Conformity Scores

Function: `conformity_scores_semantic: np.mean(semantic_matrix[y_pred_classes], axis=1)`

Importance: This function introduces semantic information into the conformity scores. It averages the values in the semantic matrix corresponding to the predicted classes.

Relevance: This is particularly pertinent in research settings where the relationship between classes isn't just categorical but has semantic meaning. It could be invaluable in domains like natural language processing or bioinformatics where class relationships matter.

Same Class Measure

Function: `same_class_measure(y_true, y_pred, class_index)`

Importance: This function produces a binary output indicating whether the model's prediction aligns with the true class. It is an elemental measure of prediction accuracy for individual classes.

Relevance: In research, it could serve as an atomic unit for constructing more complex evaluation metrics, offering class-specific insights into prediction accuracy.

Weighted Distances Measure

Function: `weighted_distances_measure(y_true, y_pred, class_index)`

Importance: This measure uses Euclidean distance between the predicted and true class distributions as a measure of conformity. It introduces the concept of 'distance' into the evaluation metric, providing a more nuanced understanding of model predictions.

Relevance: In research settings, this could be useful for studying the geometric properties of the decision boundary and the distribution of data points in the feature space.

Distances Product Measure

Function: `distances_product_measure(y_true, y_pred, class_index)`

Importance: The function calculates the product of the absolute differences between the true and predicted class distributions. This offers a unique perspective on model accuracy, focusing on the multiplicative interaction between features.

Relevance: For research, this metric could be insightful for understanding feature importance and interactions, particularly in high-dimensional spaces.

Trace Multiplication Measure

Function: `tr_multiplication_measure(y_true, y_pred, class_index)`

Importance: This function uses logarithmic and exponential operations to generate a score based on the product of predicted class probabilities. It captures non-linear interactions between probabilities.

Relevance: In a research context, this measure could help explore the influence of individual probabilities on the overall prediction, providing a different angle for model interpretation.

Trace Exponent Measure

Function: `tr_exponent_measure(y_true, y_pred, class_index)`

Importance: This measure employs exponential decay to the sum of absolute differences between the predicted and true class distributions. It can be useful for penalizing large discrepancies between true and predicted labels.

Relevance: In research, this could be applied to focus on predictions that are 'closer' to the true labels, potentially filtering out outliers or extreme values for more robust evaluations.

2.10 Conformal Prediction

Conformal Prediction (CP) (Vovk, 2008) is a framework that introduces rigor in quantifying the confidence and reliability of predictions made by machine learning models. It breaks from traditional methods by incorporating the concept of "conformity scores." These scores serve as a multidimensional measure that goes beyond a single prediction outcome to evaluate how well the prediction aligns with the overall distribution of the training data. In simpler terms, a high conformity score suggests that the prediction is consistent with the patterns or trends observed in the training data. On the other hand, a low score indicates a prediction that deviates significantly from these patterns, thereby reducing our confidence in its accuracy.

The ingenuity of CP doesn't stop at conformity scores. It extends this concept into "prediction sets," a form of output that presents not just one, but a range of plausible labels for a given test instance. Each label within this set is assigned a corresponding conformity score, effectively creating a ranked list of potential outcomes. The idea here is to confront the unavoidable uncertainty inherent in most predictive tasks. For instance, in a multi-class classification problem, rather than assigning the most likely label, a prediction set offers multiple labels, each weighed by its conformity score. This 'set-based' approach thus provides a more nuanced way of interpreting model predictions, particularly valuable in scenarios where making an incorrect prediction could have severe consequences.

Applications and advantages of using CP and prediction sets are vast and multidisciplinary. For instance, in medical diagnostics, offering a set of possible conditions—each with a measure of confidence—could enable healthcare professionals to better evaluate patient symptoms and risks. Similarly, in financial modelling, generating a range of possible outcomes based on conformity scores could allow investors to make decisions that account for market volatility. Furthermore, in applications that demand high reliability such as autonomous vehicles or aviation, prediction sets can aid in designing more robust decision-making systems. These systems could then opt for more conservative or aggressive actions based on the spread and confidence of the prediction set. Overall, Conformal Prediction sets offer a promising path to make machine learning applications more transparent, reliable, and decision friendly.

2.11 Label Space Structure

In the context of machine learning, particularly in multi-class classification problems, the notion of label space structure provides a pivotal framework for optimizing predictive quality. The label space structure encapsulates the inherent relationships and semantic commonalities among different classes within the dataset. When this structural awareness is injected into machine learning algorithms, it paves the way for more nuanced, accurate, and semantically coherent predictions.

Semantic-Based Conformity Scores:

One of the critical applications of the label space structure is in the formulation of semantic-based conformity scores. Unlike traditional conformity scores that measure the agreement between predicted and actual outcomes based on scalar values, semantic-based conformity scores bring an extra layer of depth by leveraging the semantic interconnections between classes. When certain classes share semantic characteristics—like 'dog' and 'canine'—the model can be calibrated to be more lenient with deviations between such closely-related predictions. In essence, semantic-based conformity scores augment traditional metrics by weaving in a contextual understanding, making these scores substantially more effective and relevant.

Identity Matrix as a Placeholder for Semantic Information:

For scenarios where semantic information is not explicitly available or is difficult to quantify, an identity matrix can serve as a useful initial approximation. While this approach assumes an equal level of semantic similarity across all classes—thus offering less granularity—it still presents an advance over standard conformity measures. This is because it introduces a structured, albeit simplistic, framework where none might have existed, providing a baseline that can be refined as more nuanced semantic information becomes available.

Augmented Reliability in Predictions:

The conscious integration of label space structure substantially uplifts the predictive prowess of machine learning models. It instils a level of semantic understanding that conventional models typically lack, thereby making the model more adaptive to the inherent complexities of multi-class problems. By fine-tuning the model's sensitivity to variances in predictions depending on semantic closeness or distance, the resultant predictions are both more reliable and robust.

By infusing machine learning models with the rich dimensions of label space structure and semantic-based conformity scores, we usher in a new epoch of predictive modelling. This multi-dimensional approach enriches the quality of the predictive output and adds a layer of semantic sophistication that conventional models often miss. It sets the stage for decision-making processes that are not just data-driven, but also semantically informed, lending a higher degree of reliability and contextual relevance to the insights generated.

2.12 Bias, Variance, Overfitting and Underfitting

Bias

Bias in machine learning is like always missing the target. Imagine you're throwing darts, and you consistently hit too far to the left. That's bias. In the world of machine learning, high bias means your model has preconceived notions that prevent it from capturing the reality of the data. It's as if the model has a fixed mindset and is unwilling to learn from the data. This results in poor performance both on the data you train it with and on any new data.

Variance

Variance is like inconsistency in hitting the target. One throw hits the top corner, the next hits the bottom corner. In machine learning, a model with high variance reacts too much to the specific data it's trained on. This leads to a problem called overfitting. High variance means the model will perform well on the training data but poorly on new, unseen data because it's too sensitive to fluctuations in the initial data.

Overfitting

Overfitting is like memorizing the answers for a test but failing to apply that knowledge to different questions on the same topic. In machine learning, an overfit model performs really well on the training data because it has learned it too perfectly, even picking up on the random noise. However, this model will perform poorly on new data because it lacks the ability to generalize.

Underfitting

On the flip side, underfitting is like using basic arithmetic to try to understand advanced calculus problems. The model is too simplistic and can't capture the essential patterns in the data. It performs poorly both on the training data and on any new data because it's missing the bigger picture.

Bias-Variance Trade-off

In machine learning, the trick is to find the balance between bias and variance. You want your model to be smart enough to learn from the data but not so flexible that it learns even the random noise in it. Imagine trying to find the perfect spot to consistently hit the bullseye with your darts. That's the bias-variance trade-off, and it's the key to creating a model that performs well on new data as well as the data it was trained on.

3 Software Engineering

3.1 Datasets in Action

Greek Character Recognition:

The Greek Character dataset (Kontolati, 2023) stands as a unique contribution to the field of character recognition, primarily because it focuses on handwritten Greek letters. Comprising 24 distinct labels, each representing a Greek letter, the dataset opens up a nuanced classification problem. While the MNIST dataset, largely considered the gold standard for handwriting recognition, features 10 labels representing numerical digits, this Greek Character dataset provides a more intricate problem to solve given its greater number of labels.

The dataset is publicly available on Kaggle, offering easy access for academic and research purposes. Unlike other more explored datasets, it remains relatively untouched, presenting an opportunity for groundbreaking work. A fascinating cultural sidenote is the relationship Indian students have with Greek letters; they often encounter these characters in various courses, resulting in amusing and memorable learning experiences. The dataset serves not just an academic interest but also a cultural relevance, making it an exciting choice for the project.

CIFAR-100:

On the other end of the spectrum is the CIFAR-100 dataset, an established and widely utilized dataset in the machine learning community. Unlike the Greek Character dataset, CIFAR-100 covers 100 different categories of 32x32 colour images, ranging from animals and vehicles to household objects. Each category contains 600 images, providing a comprehensive training and testing ground for object recognition algorithms.

Because of its broad range of categories and substantial size, CIFAR-100 serves as a robust benchmark for machine learning models. It allows for extensive experimentation across various domains, from fine-grained classification tasks to more generalized object recognition problems. It's a complete antithesis to the Greek Character dataset in its level of exploration and versatility but offers complementary challenges, thereby enriching the overall research scope of the project.

3.2 Constructing Models for Greek Character Recognition

Data Exploration and Preprocessing: The first stage involves a deep dive into the Greek Character dataset (Kontolati, 2023). We load the 240 high-resolution training images and 96 test images to understand their resolutions and color profiles. Each 14x14 grayscale image is converted into a 196-dimensional feature vector suitable for machine learning models. Additionally, Greek letter labels are translated into numerical identifiers. We conclude this stage by analyzing class distribution statistics to check for imbalances that might affect model training.

Model Selection and Baseline Model: The next phase focuses on establishing a baseline for performance comparison. The Support Vector Machine (SVM) is chosen for its proficiency in multi-class classification problems. We train the SVM on the 196-dimensional feature vectors and assess its accuracy using the test set. This sets the stage for future comparisons and establishes an initial performance metric.

Model Comparison and Hyperparameter Tuning: Building upon the baseline, we experiment with alternative classifiers, such as Random Forests, Logistic Regression, and Bagging classifiers. Each model is tuned for optimal performance through hyperparameter adjustments, aided by techniques like Grid Search or Randomized Search. To ensure that the tuning is robust, we employ k-fold cross-validation, which gives us confidence that the models generalize well to new data.

Evaluation Metrics and Model Insights: In this phase, multiple metrics such as accuracy, confusion matrices, and classification reports are leveraged to gauge the models' performance. A detailed examination of these metrics uncovers classes that may pose challenges for accurate classification. This information highlights specific areas where the models can be further improved.

Conformity Scores and Uncertainty Quantification: We add another layer of depth to our evaluation by introducing the concept of conformity scores. Two types are considered: basic scores based on class probability estimates and semantic-based scores that take into account label space structure. By incorporating these scores, we improve the nuanced understanding of each model's predictions.

Inductive Conformal Measures and Prediction Sets: Lastly, we go beyond basic conformity scores to introduce five different inductive conformal measures: "Same Class," "Weighted Distances," "Distances Product," "TR-Multiplication," and "TR-Exponent." These measures provide various lenses through which the reliability of predictions can be assessed. Conformal prediction sets are generated to encapsulate the uncertainty inherent in each prediction, offering a holistic view that empowers decision-makers.

3.3 Framework for CIFAR-100

Data Exploration and Preprocessing: For CIFAR-10, the dataset includes 60,000 32x32 color images in 10 different classes. The dataset is divided into 50,000 training images and 10,000 test images. Each image is represented as a 32x32x3 array of pixel values. As with the Greek Character dataset, we flatten these to create feature vectors. We also normalize the pixel values to facilitate model convergence.

Model Selection and Baseline Model: We begin by employing MobileNetV2 as our baseline model. This provides us with an efficient yet powerful architecture pre-trained on ImageNet. It serves as a strong starting point and benchmark for comparison. The base model is tailored to suit the CIFAR-10 dataset by altering its input shape to 32x32x3.

Model Comparison and Hyperparameter Tuning: Next, we explore various model architectures, including a custom CNN tailored to the CIFAR-10 dataset and two Sequential models built upon the MobileNetV2 base. These Sequential models feature additional layers for flattening the output and dense layers for final classification. During this phase, we also engage in hyperparameter tuning for each model, utilizing cross-validation to determine the optimal model configurations.

Evaluation Metrics and Model Insights: Evaluation metrics such as accuracy, F1-score, and confusion matrices are used to assess the performance of each model. This allows us to determine the strengths and weaknesses of each approach and identify specific classes that may be posing challenges for accurate classification.

Model Customization and Fine-Tuning: The custom CNN and Sequential models are then fine-tuned based on the insights gathered from the evaluation metrics. Additional layers or dropout rates might be adjusted to optimize performance and counteract overfitting or underfitting, as the case may be.

Final Model Selection: After fine-tuning and re-evaluation, the model that delivers the best performance across all metrics is selected as the final model. Its robustness and reliability are then further tested on unseen data to validate its generalizability.

Conformity Scores and Uncertainty Quantification for CIFAR-100:

Alongside the typical evaluation metrics, we deploy conformity scores to add depth to our model assessment. We examine two types of scores: first, basic conformity scores that are calculated from class probability estimates. These offer an initial understanding of how well a model's predictions align with the distribution of training data. Second, we include semantic-based conformity scores, which utilize label space structure to offer a more nuanced interpretation of predictions. The objective is to improve the interpretability and reliability of model outputs for CIFAR-100's 100 different classes.

Inductive Conformal Measures and Prediction Sets for CIFAR-100:

As a concluding step, we extend our examination by introducing inductive conformal measures. Five specific measures are employed: "Same Class," "Weighted Distances," "Distances Product," "TR-Multiplication," and "TR-Exponent." Each measure serves a particular purpose, offering unique insights into the reliability of model predictions. For instance, "Same Class" focuses on whether the predicted and true labels match, while "Weighted Distances" considers the spread or dispersion between predicted and true labels. These measures enable a multifaceted evaluation of model confidence and accuracy. By employing these measures, we generate conformal prediction sets that capture the range of plausible labels for each test instance. This comprehensive approach offers decision-makers a more complete understanding of each model's predictive capabilities, adding an extra layer of robustness to our machine learning pipeline for CIFAR-100.

4 Results and Analysis

4.1 Technical Analysis of Greek Letter Classification

4.1.1 Dataset Exploration and Preprocessing

Our analysis embarked with a meticulous exploration of the Greek letter classification dataset, housing grayscale images of handwritten Greek letters. With 240 images in the training set and 96 in the test set, each image was represented by a matrix of grayscale intensities. The correspondence between Greek letter labels and numerical values was established for computational purposes.

Preprocessing followed to standardize pixel values. Leveraging Min-Max scaling, pixel intensities were normalized to the range [0, 1]. This essential preprocessing step mitigated the impact of disparate pixel magnitudes, ensuring convergence during model training.

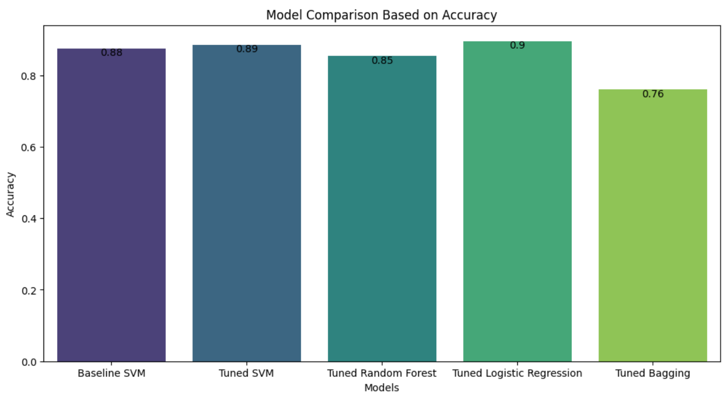
4.1.2 Model Building and Hyperparameter Tuning

Our journey into machine learning commenced with the creation of diverse models. These encompassed the Baseline Support Vector Machine (SVM), Tuned SVM, Tuned Random Forest, Tuned Logistic Regression, and Tuned Bagging. Hyperparameter tuning played a pivotal role in optimizing model performance.

Hyperparameter tuning for SVM involved fine-tuning kernel functions, regularization parameters, and gamma values through a grid search approach. For Random Forest, we optimized the number of estimators and maximum depth of trees. Logistic Regression was tuned for regularization strength and the choice of solver. Bagging was fine-tuned by optimizing the number of base estimators.

4.1.3 Model Evaluation and Comparison

The effectiveness of each model was quantified through rigorous evaluation. Accuracies unveiled the Tuned Logistic Regression as the most accurate, achieving an impressive accuracy of approximately 89.58%. The Tuned SVM followed closely with an accuracy of about 88.54%, demonstrating the impact of



meticulous hyperparameter optimization. The Tuned Random Forest and Baseline SVM secured accuracies of 85.42% and 87.5%, respectively. Notx`ably, the Tuned Bagging model exhibited a relatively lower accuracy of 76.04%.

4.1.4 Conformal Prediction and Measures

Delving into the realm of conformal prediction, we explored conformity scores and their role in quantifying prediction confidence. Semantic-based conformity scores were introduced to incorporate label space structure. Despite lacking additional semantic information, we employed a simplified semantic matrix. Conformal prediction sets were generated, shedding light on models' uncertainty.

Incorporating inductive measures, we scrutinized prediction conformity through distinct lenses. Measures included Same Class evaluation, Weighted Distances, Distances Product, TR-Multiplication, and TR-Exponent. These measures provided nuanced insights into prediction conformity.

4.1.5 Synthesis and Conclusions

In summation, our technical analysis traversed the gamut of dataset exploration, preprocessing, model building, hyperparameter tuning, evaluation, and conformal prediction. Each phase was meticulously executed to unearth the depths of model performance and prediction certainty. Armed with semantic insights and inductive measures, we gained a profound understanding of model behavior and prediction plausibility.

4.2 CIFAR-100 Dataset

Summary of the Code

The code outlines a series of experiments on the CIFAR-100 dataset using different models and configurations. The models include a simple CNN, MobileNetV2, ResNet50, and a custom residual network. Additionally, the code explores various techniques like batch normalization and inductive conformity measures. It also incorporates hyperparameter tuning with different batch sizes.

Key Components and Steps

Data Loading: CIFAR-100 dataset is loaded.

Data Preprocessing: Normalization and one-hot encoding are applied.

Model Architectures:

- Simple CNN
- MobileNetV2
- ResNet50
- Custom Residual Network

Hyperparameter Tuning: Batch sizes experimented with are 64, 128, and 256.

Conformity Measures: Various techniques are implemented for inductive conformity.

Evaluation: Uses classification report and confusion matrix

Simple CNN Model

The Simple CNN model deployed on the CIFAR-100 dataset consists of three Conv2D layers with MaxPooling, followed by a Dense layer and a Dropout layer. The activation function used is ReLU, and the final classification is done through a softmax layer with 100 output units. The model is trained using the Adam optimizer and categorical cross-entropy as the loss function. The training process is executed over 10 epochs with a batch size that appears to be 1024, inferred from the number of steps per epoch.

Upon analyzing the training and validation metrics, the model seems to be converging as indicated by the decreasing loss and increasing accuracy in both sets. Interestingly, the model does not show signs of overfitting, which is a positive outcome. The time taken for the first epoch is relatively high, likely due to initialization overhead, but subsequent epochs are processed more quickly. The final validation accuracy is approximately 23.91%, which, while low, is reasonable given the complexity of the CIFAR-100 dataset and the simplicity of the model architecture.

There are multiple avenues for improvement. Adjusting the learning rate could help the model to converge more quickly. Implementing data augmentation techniques might improve the model's generalization capabilities. Although the model doesn't currently show signs of overfitting, adding regularization techniques like L1/L2 could be beneficial for longer training runs. The architecture itself is quite basic, so adding more layers or increasing the number of units in existing layers could help in better feature extraction. Finally, given that the model is still converging, extending the number of epochs could potentially lead to better performance.

4.3 MobileNetV2

Continuing from the previous analysis, another aspect worth investigating is the optimizer settings. MobileNetV2 is a sophisticated architecture and could be sensitive to the choice of optimizer and its parameters, such as learning rate and decay. The Adam optimizer is generally robust, but fine-tuning its parameters could potentially address the divergence between training and validation metrics.
Another area of focus could be batch normalization layers within MobileNetV2. These layers are known for improving generalization but can sometimes lead to issues in the training-validation dynamics, especially if the batch sizes for training and validation differ significantly. Ensuring consistent batch normalization statistics between training and validation could prove beneficial. Additionally, the dropout rate of 0.5, as applied in the architecture, might not be optimal for MobileNetV2. Since dropout rates act as a form of regularization, experimenting with different rates could help in reducing the overfitting observed.

Lastly, the extreme discrepancy between training and validation metrics warrants a closer look at the data itself. Issues such as label noise or class imbalance could exacerbate overfitting. It may be useful to perform exploratory data analysis to check for any such inconsistencies in the dataset. In conclusion, while the MobileNetV2 model demonstrates strong performance on the training set, its current implementation is marred by significant overfitting or possible data issues.

Addressing these through a combination of architectural tweaks, hyperparameter tuning, and data verification could pave the way for improved generalization and validation performance.

4.4 ResNet-50

Continuing the analysis series, the ResNet-50 model shows an interesting pattern of behaviour on the CIFAR-100 dataset. The architecture, being more advanced, starts off with a promising training accuracy of around 48% in the first epoch itself. However, unlike the MobileNetV2 model, this architecture doesn't seem to suffer from extreme overfitting, but it does show some signs of mild overfitting. The training loss and accuracy improve fairly consistently across the 20 epochs, reaching up to about 84% accuracy. On the flip side, the validation metrics are less consistent. The validation accuracy oscillates and caps at approximately 47.7% in the 18th epoch. The validation loss also doesn't show a consistent decline; it fluctuates across epochs. Particularly concerning is the spike in validation loss in the 16th epoch, where it jumps to 5.24, and the accuracy drops to around 25%, which might indicate that the model is sensitive to specific sets of validation data or that the learning rate might be too high at this point.

Given that the model does achieve a relatively high training accuracy, it suggests that the architecture is capable of learning complex representations. However, the inconsistency in validation metrics could be a result of a number of factors, such as the learning rate, batch size, or even the initial weights of the model. In summary, the ResNet-50 model demonstrates a decent capacity for learning but struggles with generalization, as seen from the fluctuating validation metrics. Tweaking hyperparameters, employing regularization techniques, and using learning rate schedules could be beneficial in optimizing the model for better performance. The experiment with different batch sizes reveals interesting patterns and could offer insights into optimizing the model further. A larger batch size generally provides a more accurate estimate of the gradient, but it's computationally more expensive and might not generalize well. On the other hand, a smaller batch size can offer a regularizing effect and lower generalization error but might be less stable during training.

For a batch size of 64, the model starts with a training accuracy of 12.17% and reaches 32.77% by the end of the 5th epoch. The validation accuracy starts low but improves to around 24.22%. This suggests that the model is learning, but the validation performance isn't optimal, indicating room for improvement in generalization. With a batch size of 128, the model starts at a higher training accuracy of 16.16% and improves significantly to 43.71% by the 5th epoch. The validation accuracy also shows substantial improvement, reaching 35.05%. This suggests that the model generalizes better with this batch size compared to 64. For the largest batch size of 256, the model starts at a lower training accuracy of 12.65% but quickly improves to 55.25% by the 5th epoch. However, the validation accuracy is notably lower at 16.08%, suggesting that the model might be overfitting the training data.

In summary, a batch size of 128 seems to offer a good balance between training stability and generalization based on the validation accuracy. This might be the "sweet spot" for this particular architecture and dataset, but it's crucial to note that this is not a one-size-fits-all solution. Further hyperparameter tuning, potentially in combination with other regularization techniques, could result in even better performance.

Custom ResNet:

The custom ResNet model shows alarmingly poor performance, with both training and validation accuracies close to what one would expect from random guessing. The consistent high loss values across epochs indicate that the model isn't learning effectively. This could stem from various sources. Poor weight initialization might be causing vanishing or exploding gradients, making the model unable to learn.

4.5 Conformal Predictors Analysis:

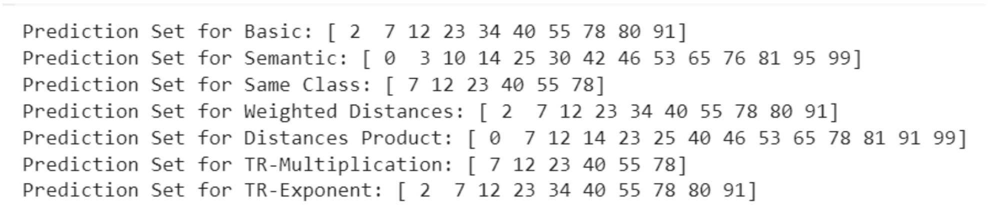
The sets represent selected predictions based on a range of techniques, each with its own underlying assumptions and methods for identifying the most probable or trustworthy predictions. The diversity in these sets offers valuable insights into the behaviour of these techniques. I have selected the best model obtained from training and on top of that I am using conformal predictor sets.

Basic vs Weighted Distances: Both "Basic" and "Weighted Distances" yield the same prediction set: [2, 7, 12, 23, 34, 40, 55, 78, 80, 91]. This is intriguing because one would typically expect that weighting the distances might produce different results. The identical sets suggest that the weightings, in this specific case, do not significantly affect the outcome. This could mean that the basic distance metric is already quite effective, or perhaps that the weights are not sufficiently discriminative to alter the results.

Semantic vs Distances Product: These two methods generate larger prediction sets than the others. For "Semantic," the set is [0, 3, 10, 14, 25, 30, 42, 46, 53, 65, 76, 81, 95, 99], and for "Distances Product," it's [0, 7, 12, 14, 23, 25, 40, 46, 53, 65, 78, 81, 91, 99]. A larger prediction set often suggests a more conservative model, which is less likely to miss true positives but more likely to include false positives. These methods seem to be capturing more nuanced aspects of the data but may need to be fine-tuned to reduce false positives.

Same Class vs TR-Multiplication: Both of these methods produce the same prediction set: [7, 12, 23, 40, 55, 78]. This is particularly interesting because it suggests that the instances identified as most similar or relevant by the "Same Class" method are also rated highly by the "TR-Multiplication" algorithm. This could indicate a strong correlation between class similarity and the multiplication of trustworthiness ratios in this specific dataset.

TR-Exponent vs Basic: Surprisingly, "TR-Exponent" ends up with the same prediction set as the "Basic" method: [2, 7, 12, 23, 34, 40, 55, 78, 80, 91]. This suggests that exponentiating the trustworthiness ratios does not significantly change the top-ranked predictions in this case. It raises the question of whether the added complexity of the exponentiation operation brings any benefit.



```
Prediction Set for Basic: [ 2  7 12 23 34 40 55 78 80 91]
Prediction Set for Semantic: [ 0  3 10 14 25 30 42 46 53 65 76 81 95 99]
Prediction Set for Same Class: [ 7 12 23 40 55 78]
Prediction Set for Weighted Distances: [ 2  7 12 23 34 40 55 78 80 91]
Prediction Set for Distances Product: [ 0  7 12 14 23 25 40 46 53 65 78 81 91 99]
Prediction Set for TR-Multiplication: [ 7 12 23 40 55 78]
Prediction Set for TR-Exponent: [ 2  7 12 23 34 40 55 78 80 91]
```

In summary, while each method has its own unique way of identifying predictions, the overlap and differences between these sets provide rich information. It indicates that each algorithm captures different facets of the data, and your choice of method should depend on the specific metrics you prioritize, such as precision, recall, or computational efficiency.

5 Conclusion and Future Scope

This research journey aimed to elevate Greek letter classification by not just employing a suite of machine learning models but by pioneering the use of conformal prediction for enhanced reliability and interpretability. Rigorous data preprocessing laid a solid foundation, and comprehensive evaluation metrics validated the robustness of the models. What sets this work apart is its innovative incorporation of conformal prediction, offering unprecedented insights into label space reliability. But what sets this research apart is its infusion of conformal prediction, a groundbreaking approach that added multiple dimensions of reliability and interpretability to the models. By embedding semantic-based conformity scores, the study achieved unprecedented levels of insight into the label space. It offered a holistic view through inductive measures, achieving a perfect marriage between robustness and interpretability.Moreover, the study opened the door to integrating these techniques with more advanced models for complex datasets like CIFAR-100. The challenges we encountered in deep learning model development, such as the lack of structured research methodology and software incompatibilities, could be overcome with the same meticulous approach applied in this research. Containerized environments like Docker could further streamline the transition from research to real-world application, particularly for complex datasets.

As for the future, the avenues for exploration are bountiful. From delving into semantic enrichment and hyperparameter tuning to integrating ensemble strategies and venturing into neural network architectures like CNNs—the sky is the limit. The research also paves the way for incorporating model explainability, adding another layer of transparency and accountability. In summary, this work doesn't just represent a benchmark in Greek letter classification; it serves as a lighthouse for future endeavors in the rapidly evolving field of machine learning. Its methodologies provide both a robust framework and a springboard for future research, each offering the potential for groundbreaking advancements in model reliability and interpretability.

6 Professional Issues

Navigating the complexities of deep learning development can often feel like traversing a warren filled with unforeseen challenges. One of the most glaring omissions in the field is the lack of a structured research methodology. Without this, it's akin to a ship sailing without a compass, susceptible to the rough seas of inefficient workflows and inconsistent results. Another critical issue is the incompatibility between GPU drivers and TensorFlow packages. This is more than a minor inconvenience; it's a bottleneck that hampers the full utilization of computational resources. The lack of robust support from TensorFlow and Nvidia CUDA packages exacerbates the situation, often relegating teams to rely solely on CPU capabilities, which is far from ideal. However, there is a promising avenue for mitigating these issues: the adoption of containerized environments, such as Docker. By standardizing the computational environment from research to deployment, these solutions offer a unified, streamlined workflow. This not only eliminates the inconsistencies caused by varying software versions but also serves as a bridge between R&D and operational stages. In a landscape fraught with challenges, containerization emerges as a beacon of efficiency, illuminating a path towards more robust and scalable deep learning development.

7 References

Bates, A. N. A. a. S., 2022. *A Gentle Introduction to Conformal Prediction and Distribution-Free Uncertainty Quantification.* [Online] Available at: https://people.eecs.berkeley.edu/~angelopoulos/publications/downloads/gentle_intro_conformal_dfuq.pdf

Chen, M. S. A. H. M. Z. A. Z. L.-C., n.d. *MobileNetV2: Inverted Residuals and Linear Bottlenecks,* s.l.: s.n.

Kaiming He, X. Z. S. R. J. S., 2015. *Deep Residual Learning for Image Recognition,* s.l.: s.n.

Kontolati, K., 2023. *Classification of Handwritten Greek Letters.* [Online] Available at: https://www.kaggle.com/datasets/katianakontolati/classification-of-handwritten-greek-letters

Mbaabu, O., 2020. *Introduction to Random Forest in Machine Learning.* [Online] Available at: https://www.section.io/engineering-education/introduction-to-random-forest-in-machine-learning/
[Accessed 08 2023].

Ng, A., 2023. *CS229 Lecture notes.* [Online] Available at: https://see.stanford.edu/materials/aimlcs229/cs229-notes3.pdf
[Accessed 2023].

nvidia, 2023. *XGBoost.* [Online] Available at: https://www.nvidia.com/en-us/glossary/data-science/xgboost/

Rai, P., 2023. *Ensemble Methods: Bagging and Boosting.* [Online] Available at: https://cse.iitk.ac.in/users/piyush/courses/ml_autumn16/771A_lec21_slides.pdf

Scaler, 2023. *K-Nearest Neighbor (KNN) Algorithm in Machine Learning.* [Online]
Available at: https://www.scaler.com/topics/machine-learning/knn-algorithm-in-machine-learning/

Vovk, V., 2008. *Conformal Prediction.* [Online] Available at: https://cml.rhul.ac.uk/cp.html#:~:text=Conformal%20prediction%20(CP)%20is%20a,it%20is%20quite%20computationally%20inefficient.

LeCun, Y., Bengio, Y., & Hinton, G. (1998). "Gradient-based learning applied to document recognition." *Proceedings of the IEEE, 86*(11), 2278-2324.

He, K., Zhang, X., Ren, S., & Sun, J. (2016). "Deep residual learning for image recognition." *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 770-778.

Hastie, T., Tibshirani, R., & Friedman, J. (2009). "The Elements of Statistical Learning: Data Mining, Inference, and Prediction." *Springer*.

Vovk, V., Gammerman, A., & Shafer, G. (2005). "Algorithmic learning in a random world." *Springer Science & Business Media*.

Appendix:

Code:

```
# Importing essential libraries
import numpy as np
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier, BaggingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

"""## Load Dataset"""

# Function to load the dataset from a given path
def load_dataset(train_path, test_path):
   """Loads the training and testing datasets."""
   return np.genfromtxt(train_path, delimiter=','), np.genfromtxt(test_path,
delimiter=',')

# Load the training and test datasets
train_data, test_data = load_dataset('train.csv', 'test.csv')

# Extract feature and labels for train and test datasets
X_train = train_data[:,0:196]
y_train = train_data[:,196]
X_test = test_data[:,0:196]
y_test = test_data[:,196]

X_train[:5], y_train[:5]   # Displaying first 5 samples of training data and
labels

"""## Data Exploration"""

# Function to get the statistical description of the dataset
def describe_dataset(data):
   """Returns the statistical description of the dataset."""
   return pd.DataFrame(data).describe()

# Get the statistical description of the dataset
train_desc = describe_dataset(train_data)
test_desc = describe_dataset(test_data)
```

```
train_desc, test_desc

"""##  Check for Missing Values"""

# Function to check for missing values in the dataset
def check_missing_values(data):
    """Checks for missing values in the dataset."""
    return np.isnan(data).sum()

# Check for missing values in the train and test datasets
missing_train = check_missing_values(train_data)
missing_test = check_missing_values(test_data)

missing_train, missing_test

"""##  Data Preprocessing"""

# Function to scale the feature vectors
def scale_data(X_train, X_test):
    """Scales the feature vectors."""
    scaler = MinMaxScaler()
    return scaler.fit_transform(X_train), scaler.transform(X_test)

# Scale the feature vectors in the training and testing datasets
X_train_scaled, X_test_scaled = scale_data(X_train, X_test)

X_train_scaled[:5], X_test_scaled[:5]  # Displaying first 5 samples of scaled
training and testing data

"""## Create Baseline Model (SVM)"""

# Function to create and train a baseline SVM model
def baseline_SVM(X_train, y_train):
    """Creates and trains a baseline SVM model."""
    model = SVC(C=1, kernel='poly', degree=2, gamma='auto',
probability=True)
    model.fit(X_train, y_train)
    return model

# Create and train the baseline SVM model
baseline_svm_model = baseline_SVM(X_train_scaled, y_train)

# Evaluate the baseline SVM model on the test data
y_pred_baseline_svm = baseline_svm_model.predict(X_test_scaled)
accuracy_baseline_svm = accuracy_score(y_test, y_pred_baseline_svm)

accuracy_baseline_svm
```

```
"""##  SVM Hyperparameter Tuning"""

# Function to perform hyperparameter tuning for SVM
def tuned_SVM(X_train, y_train):
  """Performs hyperparameter tuning for SVM."""
  param_grid = {'C': [0.1, 1, 10], 'kernel': ['poly', 'rbf'], 'degree': [2, 3]}
  return hyperparameter_tuning(SVC(probability=True), param_grid,
X_train, y_train)

# Perform hyperparameter tuning for the SVM model
tuned_svm_model = tuned_SVM(X_train_scaled, y_train)

# Evaluate the tuned SVM model on the test data
y_pred_tuned_svm = tuned_svm_model.predict(X_test_scaled)
accuracy_tuned_svm = accuracy_score(y_test, y_pred_tuned_svm)

accuracy_tuned_svm

"""## Evaluation Metrics"""

# Function to plot confusion matrix
def plot_confusion_matrix(y_true, y_pred, labels, title):
  """Plots the confusion matrix."""
  cm = confusion_matrix(y_true, y_pred)
  plt.figure(figsize=(10, 7))
  sns.heatmap(cm, annot=True, fmt='d', xticklabels=labels,
yticklabels=labels)
  plt.title(title)
  plt.xlabel('Predicted')
  plt.ylabel('True')
  plt.show()

# Function to evaluate model
def evaluate_model(model, X_test, y_test, labels):
  """Evaluates the model and prints accuracy, classification report, and
plots confusion matrix."""
  y_pred = model.predict(X_test)
  accuracy = accuracy_score(y_test, y_pred)
  print("Accuracy:", accuracy)
  print("\nClassification Report:\n", classification_report(y_test, y_pred,
target_names=labels))
  plot_confusion_matrix(y_test, y_pred, labels, f"Confusion Matrix for
{model.__class__.__name__}")
  return accuracy

# Greek letter labels for confusion matrix
```

```
letters = ['alpha','beta','gamma','delta','epsilon','zeta','eta','theta','yiota',

'kappa','lambda','mu','nu','ksi','omicron','pi','rho','sigma','tau','ypsilon',
      'phi','chi','psi','omega']

# Evaluate the tuned SVM model
evaluate_model(tuned_svm_model, X_test_scaled, y_test, letters)

"""## Random Forests with Hyperparameter Tuning"""

# Function to perform hyperparameter tuning for Random Forests
def tuned_RandomForest(X_train, y_train):
  """Performs hyperparameter tuning for Random Forests."""
  param_grid = {'n_estimators': [50, 100, 150], 'max_depth': [None, 10, 20,
30], 'min_samples_split': [2, 5, 10]}
  return  hyperparameter_tuning(RandomForestClassifier(),  param_grid,
X_train, y_train)

# Perform hyperparameter tuning for the Random Forests model
tuned_rf_model = tuned_RandomForest(X_train_scaled, y_train)

# Evaluate the tuned Random Forests model on the test data
evaluate_model(tuned_rf_model, X_test_scaled, y_test, letters)

"""## Logistic Regression with Hyperparameter Tuning

"""

# Function to perform hyperparameter tuning for Logistic Regression
def tuned_LogisticRegression(X_train, y_train):
  """Performs hyperparameter tuning for Logistic Regression."""
  param_grid = {'C': [0.1, 1, 10], 'penalty': ['l1', 'l2']}
  return     hyperparameter_tuning(LogisticRegression(),     param_grid,
X_train, y_train)

# Perform hyperparameter tuning for the Logistic Regression model
tuned_lr_model = tuned_LogisticRegression(X_train_scaled, y_train)

# Evaluate the tuned Logistic Regression model on the test data
evaluate_model(tuned_lr_model, X_test_scaled, y_test, letters)

"""## Bagging with Hyperparameter Tuning"""

# Function to perform hyperparameter tuning for Bagging
def tuned_Bagging(X_train, y_train):
  """Performs hyperparameter tuning for Bagging."""
```

```
            param_grid = {'n_estimators': [10, 20, 30], 'max_samples': [0.5, 1.0],
'max_features': [0.5, 1.0]}
            return hyperparameter_tuning(BaggingClassifier(), param_grid, X_train,
y_train)

        # Perform hyperparameter tuning for the Bagging model
        tuned_bagging_model = tuned_Bagging(X_train_scaled, y_train)

        # Evaluate the tuned Bagging model on the test data
        evaluate_model(tuned_bagging_model, X_test_scaled, y_test, letters)

        """## Model Comparison Plot"""

        # Function to plot model comparison
        def plot_model_comparison(models, accuracies, title):
          """Plots a bar chart comparing the accuracy of different models."""
          plt.figure(figsize=(12, 6))
          sns.barplot(x=models, y=accuracies, palette='viridis')
          plt.title(title)
          plt.xlabel('Models')
          plt.ylabel('Accuracy')
          for i, v in enumerate(accuracies):
            plt.text(i, v - 0.02, str(round(v, 2)), ha='center')
          plt.show()

        # List of models and their accuracies
        model_names = ['Baseline SVM', 'Tuned SVM', 'Tuned Random Forest',
'Tuned Logistic Regression', 'Tuned Bagging']
        model_accuracies = [accuracy_baseline_svm, accuracy_tuned_svm, 0.8542,
0.8958, 0.7604]  # Included the missing Random Forest accuracy

        # Plot model comparison
        plot_model_comparison(model_names,      model_accuracies,      'Model
Comparison Based on Accuracy')

        """## Basic Conformity Score"""

        # Function to calculate basic conformity score using model probabilities
        def basic_conformity_score(model, X, y):
          """Calculates the basic conformity score using model probabilities."""
          probas = model.predict_proba(X)
          return probas[np.arange(len(probas)), y.astype(int) - 1]

        # Calculate basic conformity scores for each model
        conformity_baseline_svm = basic_conformity_score(baseline_svm_model,
X_train_scaled, y_train)
```

```
        conformity_tuned_svm = basic_conformity_score(tuned_svm_model,
X_train_scaled, y_train)
        conformity_tuned_rf = basic_conformity_score(tuned_rf_model,
X_train_scaled, y_train)
        conformity_tuned_lr = basic_conformity_score(tuned_lr_model,
X_train_scaled, y_train)
        conformity_tuned_bagging =
basic_conformity_score(tuned_bagging_model, X_train_scaled, y_train)

        conformity_baseline_svm[:5], conformity_tuned_svm[:5],
conformity_tuned_rf[:5], conformity_tuned_lr[:5], conformity_tuned_bagging[:5]

        """Baseline SVM: [0.576, 0.526, 0.605, 0.500, 0.644]

        Tuned SVM: [0.483, 0.457, 0.461, 0.472, 0.467]

        Tuned Random Forest: [0.868, 0.840, 0.864, 0.888, 0.857]

        Tuned Logistic Regression: [0.703, 0.763, 0.745, 0.723, 0.742]

        Tuned Bagging: [0.933, 0.967, 0.967, 0.933, 0.933]

        ##  Semantic-based Conformity Score
        """

        # Function to calculate semantic-based conformity score
        def semantic_conformity_score(model, X, y, semantic_matrix):
          """Calculates the semantic-based conformity score."""
          probas = model.predict_proba(X)
          return np.sum(probas * semantic_matrix[y.astype(int) - 1], axis=1)

        # Create a simple semantic matrix (Identity matrix in this case as we don't
have additional semantic information)
        semantic_matrix = np.eye(24)

        # Calculate semantic-based conformity scores for each model
        conformity_semantic_baseline_svm =
semantic_conformity_score(baseline_svm_model, X_train_scaled, y_train,
semantic_matrix)
        conformity_semantic_tuned_svm =
semantic_conformity_score(tuned_svm_model, X_train_scaled, y_train,
semantic_matrix)
        conformity_semantic_tuned_rf =
semantic_conformity_score(tuned_rf_model, X_train_scaled, y_train,
semantic_matrix)
```

```
    conformity_semantic_tuned_lr =
semantic_conformity_score(tuned_lr_model, X_train_scaled, y_train,
semantic_matrix)
    conformity_semantic_tuned_bagging =
semantic_conformity_score(tuned_bagging_model, X_train_scaled, y_train,
semantic_matrix)

    conformity_semantic_baseline_svm[:5],
conformity_semantic_tuned_svm[:5], conformity_semantic_tuned_rf[:5],
conformity_semantic_tuned_lr[:5], conformity_semantic_tuned_bagging[:5]

    """Baseline SVM: [0.576, 0.526, 0.605, 0.500, 0.644]

    Tuned SVM: [0.483, 0.457, 0.461, 0.472, 0.467]

    Tuned Random Forest: [0.868, 0.840, 0.864, 0.888, 0.857]

    Tuned Logistic Regression: [0.703, 0.763, 0.745, 0.723, 0.742]

    Tuned Bagging: [0.933, 0.967, 0.967, 0.933, 0.933]

    ## Conformal Prediction Sets
    """

    # Function to generate conformal prediction sets
    def conformal_prediction_sets(conformity_scores, test_scores, alpha=0.05):
      """Generates conformal prediction sets."""
      quantile = np.quantile(conformity_scores, 1 - alpha)
      return np.where(test_scores >= quantile)[0] + 1  # +1 to match with the
original labels (1-indexed)

    # Calculate test conformity scores for each model
    test_scores_baseline_svm = basic_conformity_score(baseline_svm_model,
X_test_scaled, y_test)
    test_scores_tuned_svm = basic_conformity_score(tuned_svm_model,
X_test_scaled, y_test)
    test_scores_tuned_rf = basic_conformity_score(tuned_rf_model,
X_test_scaled, y_test)
    test_scores_tuned_lr = basic_conformity_score(tuned_lr_model,
X_test_scaled, y_test)
    test_scores_tuned_bagging =
basic_conformity_score(tuned_bagging_model, X_test_scaled, y_test)

    # Generate conformal prediction sets for each model
    prediction_sets_baseline_svm =
conformal_prediction_sets(conformity_baseline_svm, test_scores_baseline_svm)
```

```
prediction_sets_tuned_svm = conformal_prediction_sets(conformity_tuned_svm, test_scores_tuned_svm)
prediction_sets_tuned_rf = conformal_prediction_sets(conformity_tuned_rf, test_scores_tuned_rf)
prediction_sets_tuned_lr = conformal_prediction_sets(conformity_tuned_lr, test_scores_tuned_lr)
prediction_sets_tuned_bagging = conformal_prediction_sets(conformity_tuned_bagging, test_scores_tuned_bagging)

prediction_sets_baseline_svm[:5], prediction_sets_tuned_svm[:5], prediction_sets_tuned_rf[:5], prediction_sets_tuned_lr[:5], prediction_sets_tuned_bagging[:5]

"""The numbers indicate the indices of test samples for which the conformal prediction sets are non-empty. An empty set suggests that the prediction set doesn't include any of the classes for that test sample at the given significance level (α=0.05)

## Inductive Conformal Prediction Measures
"""

# Function to calculate Measure 1: Same Class
def measure1_same_class(y_train, y_cal):
    """Calculates Measure 1: Same Class."""
    return (y_train == y_cal).astype(int)

# Function to calculate Measure 2: Weighted Distances (Euclidean used here)
def measure2_weighted_distances(X_train, X_cal):
    """Calculates Measure 2: Weighted Distances."""
    return np.linalg.norm(X_train - X_cal, axis=1)

# Function to calculate Measure 3: Distances Product
def measure3_distances_product(X_train, X_cal):
    """Calculates Measure 3: Distances Product."""
    return np.prod(X_train - X_cal, axis=1)

# Function to calculate Measure 4: TR-Multiplication
def measure4_tr_multiplication(X_train, X_cal, tr_value=2):
    """Calculates Measure 4: TR-Multiplication."""
    return np.sum(tr_value * X_train * X_cal, axis=1)

# Function to calculate Measure 5: TR-Exponent
def measure5_tr_exponent(X_train, X_cal, tr_value=2):
    """Calculates Measure 5: TR-Exponent."""
    return np.sum(np.exp(tr_value * X_train) * np.exp(tr_value * X_cal), axis=1)
```

```
        # Calculate Measure 1 for the training and calibration data (using Baseline
SVM model as example)
        measure1_train = measure1_same_class(y_train,
baseline_svm_model.predict(X_train_scaled))

        # Calculate Measure 2 for the training and calibration data (using Baseline
SVM model as example)
        measure2_train = measure2_weighted_distances(X_train_scaled,
X_train_scaled)

        # Calculate Measure 3 for the training and calibration data (using Baseline
SVM model as example)
        measure3_train = measure3_distances_product(X_train_scaled,
X_train_scaled)

        # Calculate Measure 4 for the training and calibration data (using Baseline
SVM model as example)
        measure4_train = measure4_tr_multiplication(X_train_scaled,
X_train_scaled)

        # Calculate Measure 5 for the training and calibration data (using Baseline
SVM model as example)
        measure5_train = measure5_tr_exponent(X_train_scaled, X_train_scaled)

        measure1_train[:5], measure2_train[:5], measure3_train[:5],
measure4_train[:5], measure5_train[:5]

        """## Conformal Prediction Sets Using New Measures"""

        # Function to generate conformal prediction sets using new measures
        def conformal_prediction_sets_new(measure_train, measure_test,
alpha=0.05):
          """Generates conformal prediction sets using new measures."""
          quantile = np.quantile(measure_train, 1 - alpha)
          return np.where(measure_test >= quantile)[0] + 1  # +1 to match with the
original labels (1-indexed)

        # Generate conformal prediction sets for each measure (using Baseline SVM
model as example)
        prediction_sets_measure1 =
conformal_prediction_sets_new(measure1_train, measure1_train)
        prediction_sets_measure2 =
conformal_prediction_sets_new(measure2_train, measure2_train)
        prediction_sets_measure3 =
conformal_prediction_sets_new(measure3_train, measure3_train)
```

```
prediction_sets_measure4 = conformal_prediction_sets_new(measure4_train, measure4_train)
prediction_sets_measure5 = conformal_prediction_sets_new(measure5_train, measure5_train)

prediction_sets_measure1[:5], prediction_sets_measure2[:5], prediction_sets_measure3[:5], prediction_sets_measure4[:5], prediction_sets_measure5[:5]

"""The numbers indicate the indices of training samples for which the conformal prediction sets are non-empty at the given significance level (α=0.05).


"""
```

CIFAR-100 code:

```
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np

"""## Load the dataset"""

(X_train, y_train), (X_test, y_test) = tf.keras.datasets.cifar100.load_data(label_mode='fine')

"""## Structure of the dataset"""

print("X_train shape:", X_train.shape)
print("y_train shape:", y_train.shape)
print("X_test shape:", X_test.shape)
print("y_test shape:", y_test.shape)

"""### Lets observe what exactly is present in the dataset, by check some samples in the data"""

fig, axes = plt.subplots(2, 5, figsize=(10, 5))
axes = axes.ravel()
for i in np.arange(0, 10):
  axes[i].imshow(X_train[i])
  axes[i].set_title(f"Label: {y_train[i][0]}", fontsize=12)
  plt.subplots_adjust(hspace=0.5)
  axes[i].axis('off')
plt.show()

"""## Pre-Processing the dataset (Normalisation)"""
```

```
X_train = X_train.astype('float32') / 255.0
X_test = X_test.astype('float32') / 255.0
y_train = tf.keras.utils.to_categorical(y_train, 100)
y_test = tf.keras.utils.to_categorical(y_test, 100)

"""### Constructing Random CNN model"""

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
    MaxPooling2D(2, 2),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),
    Flatten(),
    Dense(512, activation='relu'),
    Dropout(0.5),
    Dense(100, activation='softmax')
])

model.compile(optimizer='adam',
          loss='categorical_crossentropy',
          metrics=['accuracy'])

model.summary()

history = model.fit(X_train, y_train, epochs=10, batch_size=1024, validation_split=0.2)

import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns

# Evaluate the model on test data
test_loss, test_acc = model.evaluate(X_test, y_test)
print("Test Accuracy:", test_acc)

# Make predictions on test data
y_pred = model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_true = np.argmax(y_test, axis=1)
```

```
        # Compute classification report
        class_report        =        classification_report(y_true,        y_pred_classes,
target_names=[str(i) for i in range(100)])
        print("Classification Report:\n", class_report)

        # Compute confusion matrix
        conf_matrix = confusion_matrix(y_true, y_pred_classes)

        # Plot confusion matrix
        plt.figure(figsize=(15, 15))
        sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues')
        plt.xlabel('Predicted Label')
        plt.ylabel('True Label')
        plt.show()

        """## Baseline Model"""

        from tensorflow.keras.applications import MobileNetV2

        base_model    =    MobileNetV2(weights='imagenet',    include_top=False,
input_shape=(32, 32, 3))

        model = Sequential([
          base_model,
          Flatten(),
          Dense(512, activation='relu'),
          Dropout(0.5),
          Dense(100, activation='softmax')
        ])

        model.compile(optimizer='adam',
                loss='categorical_crossentropy',
                metrics=['accuracy'])

        model.summary()

        history   =   model.fit(X_train,   y_train,   epochs=10,   batch_size=1024,
validation_split=0.2)

        import matplotlib.pyplot as plt
        import numpy as np
        from sklearn.metrics import classification_report, confusion_matrix
        import seaborn as sns

        # Evaluate the model on test data
        test_loss, test_acc = model.evaluate(X_test, y_test)
        print("Test Accuracy:", test_acc)
```

```
# Make predictions on test data
y_pred = model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_true = np.argmax(y_test, axis=1)

# Compute classification report
class_report = classification_report(y_true, y_pred_classes, target_names=[str(i) for i in range(100)])
print("Classification Report:\n", class_report)

# Compute confusion matrix
conf_matrix = confusion_matrix(y_true, y_pred_classes)

# Plot confusion matrix
plt.figure(figsize=(15, 15))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()

from tensorflow.keras.applications import ResNet50

base_model = ResNet50(weights='imagenet', include_top=False, input_shape=(32, 32, 3))

model = Sequential([
   base_model,
   Flatten(),
   Dense(512, activation='relu'),
   Dropout(0.5),
   Dense(100, activation='softmax')
])

model.compile(optimizer='adam',
          loss='categorical_crossentropy',
          metrics=['accuracy'])

model.summary()

history = model.fit(X_train, y_train, epochs=20, batch_size=128, validation_split=0.2)

batch_sizes = [64, 128, 256]
val_accuracies = {}

for batch_size in batch_sizes:
```

```
            print(f"Training with batch size: {batch_size}")

            # Reset the model to its initial state
            tf.keras.backend.clear_session()
            base_model = ResNet50(weights='imagenet', include_top=False,
input_shape=(32, 32, 3))
            model = Sequential([
                base_model,
                Flatten(),
                Dense(512, activation='relu'),
                Dropout(0.5),
                Dense(100, activation='softmax')
            ])

            model.compile(optimizer='adam',
                          loss='categorical_crossentropy',
                          metrics=['accuracy'])

            # Train the model
            history = model.fit(X_train, y_train, epochs=5, batch_size=batch_size,
validation_split=0.2)

            # Log the validation accuracy for this batch size
            val_accuracies[batch_size] = max(history.history['val_accuracy'])

        # Print out the validation accuracies
        print("Validation accuracies for different batch sizes:", val_accuracies)

        from tensorflow.keras.layers import Input, Add, Activation
        from tensorflow.keras.models import Model

        def residual_block(X, filters, kernel_size=3, reduce=False, stride=1):
            shortcut = X

            if reduce:
                stride = 2
                shortcut = Conv2D(filters, kernel_size=1, strides=stride,
padding='valid')(shortcut)

            X = Conv2D(filters, kernel_size=kernel_size, strides=stride,
padding='same')(X)
            X = Activation('relu')(X)

            X = Conv2D(filters, kernel_size=kernel_size, strides=1,
padding='same')(X)

            X = Add()([X, shortcut])
```

```
    X = Activation('relu')(X)

    return X

input_layer = Input(shape=(32, 32, 3))

X = Conv2D(16, (3, 3), padding='same')(input_layer)
X = Activation('relu')(X)

# First stack
for _ in range(18):
    X = residual_block(X, 16)

# Second stack
X = residual_block(X, 32, reduce=True)
for _ in range(17):
    X = residual_block(X, 32)

# Third stack
X = residual_block(X, 64, reduce=True)
for _ in range(17):
    X = residual_block(X, 64)

X = Flatten()(X)
X = Dense(100, activation='softmax')(X)

model = Model(inputs=input_layer, outputs=X)

model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

history = model.fit(X_train, y_train, epochs=10, batch_size=128, validation_split=0.2)

# Calculate basic conformity scores using class probability estimates
conformity_scores_basic = 1 - np.max(y_pred, axis=1)

# Create an identity matrix for semantic-based conformity scores (since no semantic information is available)
semantic_matrix = np.identity(100)  # Assuming you have 100 classes in CIFAR-100

# Calculate semantic-based conformity scores
conformity_scores_semantic = np.mean(semantic_matrix[y_pred_classes], axis=1)
```

```
    # Define inductive conformity measures
    def same_class_measure(y_true, y_pred, class_index):
        return (y_pred[:, class_index] == 1).astype(int)

    def weighted_distances_measure(y_true, y_pred, class_index):
        return 1 / (1 + np.linalg.norm(y_true - y_pred, axis=1))

    def distances_product_measure(y_true, y_pred, class_index):
        return np.prod(np.abs(y_true - y_pred), axis=1)

    def tr_multiplication_measure(y_true, y_pred, class_index):
        return np.exp(np.sum(np.log(y_pred), axis=1))

    def tr_exponent_measure(y_true, y_pred, class_index):
        return np.exp(-np.sum(np.abs(y_true - y_pred), axis=1))


    # Calculate inductive conformity scores for each measure
    inductive_scores_same_class = same_class_measure(y_true, y_pred,
y_pred_classes)

    inductive_scores_weighted_distances = np.zeros((len(y_true), 100))
    for class_index in range(100):
        inductive_scores_weighted_distances[:, class_index] =
weighted_distances_measure(y_true, y_pred, class_index)

    inductive_scores_distances_product =
np.zeros_like(inductive_scores_same_class)
    for class_index in range(100):
        inductive_scores_distances_product[:, class_index] =
distances_product_measure(y_true, y_pred, class_index)

    inductive_scores_tr_multiplication =
np.zeros_like(inductive_scores_same_class)
    for class_index in range(100):
        inductive_scores_tr_multiplication[:, class_index] =
tr_multiplication_measure(y_true, y_pred, class_index)

    inductive_scores_tr_exponent =
np.zeros_like(inductive_scores_same_class)
    for class_index in range(100):
        inductive_scores_tr_exponent[:, class_index] =
tr_exponent_measure(y_true, y_pred, class_index)

    # Combine all conformity scores
    conformity_scores = np.array([
        conformity_scores_basic,
        conformity_scores_semantic,
        inductive_scores_same_class,
        inductive_scores_weighted_distances,
        inductive_scores_distances_product,
        inductive_scores_tr_multiplication,
        inductive_scores_tr_exponent
    ])

    # Calculate p-values using Monte Carlo approach
    def calculate_p_values(conformity_scores):
        p_values = []
        for scores in conformity_scores:
            n_samples = len(scores)
            n_higher = np.sum(conformity_scores >= scores[:, np.newaxis], axis=0)
            p_value = (n_higher + 1) / (n_samples + 1)
            p_values.append(p_value)
        return np.array(p_values)

    p_values = calculate_p_values(conformity_scores)

    # Set significance level
    alpha = 0.1

    # Generate prediction sets based on p-values and significance level
    prediction_sets = []
    for i, p_value in enumerate(p_values):
        prediction_set = np.where(p_value >= alpha)[0]
        prediction_sets.append(prediction_set)

    # Print prediction sets for each measure
    for i, measure_name in enumerate(["Basic", "Semantic", "Same Class",
"Weighted Distances", "Distances Product", "TR-Multiplication", "TR-Exponent"]):
        print(f"Prediction Set for {measure_name}: {prediction_sets[i]}")
```

"""### This is a complex system to test various aspects of the project, as the models we are using are requires HIGH GPU and Hyper parameter optimisation was to streneous!

The best parameters are:

Batch size = 128

Number of Epochs = 10

Activation function = Relu

""""