Unveiling the Power of Bias Adjustment: Enhancing Predictive Precision in Imbalanced Datasets

Hyung Gyu Rho · Following

Published in Towards Data Science · 11 min read · Aug 13

Addressing class imbalance is crucial for accurate predictions in data science. This article introduces Bias Adjustment to enhance model accuracy amidst class imbalance. Explore how Bias Adjustment optimizes predictions and overcomes this challenge.

Introduction

In the realm of data science, effectively managing imbalanced datasets is crucial for precise predictions. Imbalanced datasets, characterized by significant class disparities, can lead to biased models favoring the majority class and delivering subpar performance for the minority class, especially in critical contexts like fraud detection and disease diagnosis.

This article introduces a pragmatic remedy known as Bias Adjustment. By fine-tuning the bias term within the model, it counteracts class imbalance,

bolstering the model's aptitude for accurate predictions across both majority and minority classes. The article outlines algorithms catering to binary and multi-class classification, followed by an exploration of their underlying principles. Notably, the Algorithm Explanation and Underlying Principles section rigorously establishes a theoretical link between my algorithm, oversampling, and adjusting class weights, enhancing the reader's understanding.

To substantiate the efficacy and rationale, a simulation study scrutinizes the relationship between Bias Adjustment and oversampling. Further, a practical application is employed to illustrate the implementation and tangible benefits of Bias Adjustment in credit card fraud detection.

Bias Adjustment offers a direct and impactful avenue for refining predictive modeling results in the face of class imbalance. This article provides a comprehensive grasp of the mechanism, principles, and real-world implications of Bias Adjustment, making it an invaluable tool for data scientists seeking to enhance model performance amidst imbalanced datasets.

Algorithm

The Bias Adjustment algorithm introduces a methodology to address class imbalance in binary and multi-class classification tasks. By recalibrating the bias term at each epoch, the algorithm enhances the model's capacity to handle imbalanced datasets effectively. Through adjusting the bias term, the algorithm sensitizes the model to minority class instances, thereby improving classification accuracy.

Model f(x) and its Role in Predictions

At the core of our bias adjustment algorithm is the concept of $f(x)$ — a crucial factor that guides our approach to dealing with class imbalance. $f(x)$ serves as a link between input features x and the final predictions. In binary classification, it acts as a mapping that transforms inputs into real values, aligned with the sigmoid activation for probability interpretation. In multi-class classification, $f(x)$ transforms into a set of functions, $f\_k(x)$, where each class k has its own function, working in sync with the softmax activation. This distinction is instrumental in our bias adjustment algorithm, where we use $f(x)$ to adjust bias term(s) and fine-tune sensitivity to class imbalance.

Brief Overview of the Algorithm

The algorithm's concept is straightforward: calculate the average of $f\_k(x)$ for each class k and represent this average as δk. By subtracting δk from $f\_k(x)$, we ensure that the expected value of $f\_k(x) - \delta k$ becomes 0 for every class k. Consequently, the model predicts that each class is equally likely to occur. While this provides a concise glimpse into the algorithm's rationale, it's important to note that this approach is substantiated by theoretical and mathematical foundations, which will be explored further in subsequent sections of this article.

Algorithm for Binary

Algorithm 1 Algorithm with Bias Adjustment (Binary)

Initialize early stopping patience count to 0
Initialize best metric to 0
Get the number of observations in each class from train data – use $N0$ and $N1$ to denote the number of observations from classes 0 and 1, respectively
for epoch = 1 to E **do**
  Initialize bias for each class
  $\delta 0 \leftarrow 0$
  $\delta 1 \leftarrow 0$
  for batch = 1 to B **do**
    Train model $\hat{f}(x)$
    Sum $\hat{f}(x)$ value over all observations from batch for each class and update $\delta 0$ and $\delta 1$
    $\delta 0 \leftarrow \delta 0 - \sum_{i \in \text{batch} \cap C0} \hat{f}(x_i)$
    $\delta 1 \leftarrow \delta 1 - \sum_{i \in \text{batch} \cap C1} \hat{f}(x_i)$
  end for
  $\delta \leftarrow \frac{\delta 0/N0 + \delta 1/N1}{2}$
  Calculate desired evaluation metric M using $\frac{1}{1+\exp(-\delta - \hat{f}(x))}$
  if best metric $< M$ **then**
    best metric $\leftarrow M$
    early stopping patience $\leftarrow 0$
  else
    early stopping patience count $\leftarrow$ early stopping patience count $+ 1$
  end if
  if early stopping patience count $> P$ **then**
    break
  end if
end for
Return: $\hat{f}(x)$ (trained model), δ (Bias adjust term)
Notation: E (max epochs), B (number of batches), P (patience), $C0$ (set of class 0 observations), $C1$ (set of class 1 observations)

Created by Author

Utilization for Prediction: For making predictions, apply the last calculated δ value from the algorithm. This δ value reflects cumulative adjustments made during training and serves as a basis for the final bias term in the sigmoid activation function during prediction.

For example, consider the scenario where the user intends to predict the class of a specific observation, denoted as x_i. To achieve this, the algorithm involves the calculation of the expression $\frac{1}{1+\exp(-\delta - \hat{f}(x_i))}$. This computation hinges on the cumulative adjustments reflected by δ – the value accumulated over epochs through the bias adjustment process.

Upon obtaining the calculated value, a classification decision is made. If the resulting value is greater than 0.5, the observation is classified as belonging to class 1; otherwise, it is categorized as class 0. It's important to note that this classification approach assumes a threshold of 0.5, serving as the boundary for assigning class labels.

This methodology capitalizes on the sigmoid function's property to transform the adjusted bias and model output into a probability-like score, facilitating the binary classification decision process.

Algorithm for Multi-Class

Algorithm 2 Algorithm with Bias Adjustment (Multi-class)

Initialize early stopping patience count to 0
Initialize best metric to 0
Get the number of observations in each class from train data – use Nk to denote the number of observations from class k
for epoch = 1 to E **do**
  Initialize bias for each class k
  $\forall k, \delta_k \leftarrow 0$
  for batch = 1 to B **do**
    Train model $\hat{f}(x) = \{\hat{f}_k(x)\}_{k=1}^{K}$
    Sum $\hat{f}(x)$ value over all observations from batch for each class, and update δ_k
    $\delta_k \leftarrow \delta_k - \sum_{i \in \text{batch} \cap Ck} \hat{f}(x_i)$
  end for
  Divide δ_k by Nk to get the average value
  $\delta_k \leftarrow \frac{\delta_k}{Nk}$
  Calculate desired evaluation metric M using $\frac{\delta_k + \hat{f}_k(x)}{\sum_k \exp(\delta_k + \hat{f}_k(x))}$
  if best metric $< M$ **then**
    best metric $\leftarrow M$
    early stopping patience $\leftarrow 0$
  else
    early stopping patience count $\leftarrow$ early stopping patience count $+ 1$
  end if
  if early stopping patience count $> P$ **then**
    break
  end if
end for
Return: $\hat{f}(x)$ (trained model), δ (Bias adjust term)
Notation: E (max epochs), B (number of batches), P (patience), K (number of classes), Ck (set of class k observations)

Created by Author

Utilization for Prediction: The culmination of our algorithm's training process yields a crucial element — the last calculated δk value. This δk value encapsulates the cumulative bias term adjustments that have been meticulously orchestrated during training. Its significance lies in its role as a foundational parameter for the final bias term in the softmax activation function during prediction.

Then, to predict for observation i, we perform the following steps. First, calculate predictions for each class k using the formula:

$$\frac{\exp(\delta_k + \hat{f}_k(x_i))}{\sum_k \exp(\delta_k + \hat{f}_k(x_i))},$$

where δ_k is obtained from the training process according to the above algorithm. Next, classify observation i as the class with the highest probability.

Created by Author

Algorithm Explanation and Underlying Principles

> *From oversampling to adjusting class weight, From adjusting class weight to the new algorithm*

In this section, we embark on an exploration of the Algorithm's Explanation and Underlying Principles. Our aim is to elucidate the mechanics and rationale behind the algorithm's operations, providing insights into its effectiveness in addressing class imbalance in classification tasks.

Loss Function and Imbalance

We commence our journey by delving into the heart of the algorithm, the loss function. For our initial exposition, we will examine the loss function without directly addressing the issue of class imbalance. Let us consider a binary classification problem, where Class 1 comprises 90% of the observations and Class 0 constitutes the remaining 10%. Denoting the set of observations from Class 1 as C1 and from Class 0 as C0, we take this as our starting point.

The loss function, in the absence of addressing class imbalance, takes the form:

$$L(y, f(x)) = -\sum_{i \in C1} y_i \log(\frac{1}{1+exp(-f(x_i))}) - \sum_{i \in C0} (1-y_i) \log(\frac{exp(-f(x_i))}{1+exp(-f(x_i))}).$$

Created by Author

In the pursuit of model estimation, we endeavor to minimize this loss function:

$$\hat{f}(\cdot) = \arg\min_{f(\cdot)} L(y, f(x)).$$

Created by Author

Mitigating Imbalance: Oversampling and Adjusting Class Weights

However, the crux of our endeavor lies in addressing the class imbalance predicament. To surmount this challenge, we venture into employing oversampling techniques. While various oversampling methods exist — encompassing simple oversampling, random oversampling, SMOTE, and others — our focus, for presentational clarity, narrows onto simple oversampling, with a glimpse into random oversampling.

Simple Oversampling:

A fundamental approach in our arsenal is simple oversampling, a technique where we duplicate instances of the minority class by a factor of eight to match the size of the majority class. In our illustrative example, where the minority class constitutes 10% and the majority class the remaining 90%, we duplicate the minority class observations eightfold, effectively equalizing class distribution. Denoting the set of duplicated observations as D0, this step transforms our loss function as follows:

$$
\begin{aligned}
L(y, f(x)) &= -\sum_{i \in C1} y_i \log\left(\frac{1}{1+exp(-f(x_i))}\right) - \sum_{i \in C0} (1-y_i) \log\left(\frac{exp(-f(x_i))}{1+exp(-f(x_i))}\right) - \sum_{i \in D0} (1-y_i) \log\left(\frac{exp(-f(x_i))}{1+exp(-f(x_i))}\right) \\
&= -\sum_{i \in C1} y_i \log\left(\frac{1}{1+exp(-f(x_i))}\right) - \sum_{i \in C0} (1-y_i) \log\left(\frac{exp(-f(x_i))}{1+exp(-f(x_i))}\right) - 8 \sum_{i \in C0} (1-y_i) \log\left(\frac{exp(-f(x_i))}{1+exp(-f(x_i))}\right) \\
&= -\sum_{i \in C1} y_i \log\left(\frac{1}{1+exp(-f(x_i))}\right) - 9 \sum_{i \in C0} (1-y_i) \log\left(\frac{exp(-f(x_i))}{1+exp(-f(x_i))}\right).
\end{aligned}
$$

Created by Author

This reveals a profound insight: the core principle of simple oversampling seamlessly corresponds to the notion of adjusting class weights. Duplicating the minority class eightfold effectively equates to augmenting the weight of the minority class to ninefold. Significantly, the technique of oversampling mirrors the mechanism of weight adjustment.

Random Oversampling:
A brief contemplation on random oversampling underscores a parallel observation. Random oversampling, akin to its simpler counterpart, serves as an equivalent to random adjustment of observation weights.

From Adjusting Class Weights to Adjusting Bias

A key revelation underscores the core of our approach: the essential equivalence between bias adjustment, oversampling, and weight adjustment. This insight emanates from

> *"Prentice and Pyke (1979) … have shown that, when the model contains a constant (intercept) term for each category, these constant terms are the only coefficient affected by the unequal selection probability of the Y" Scott & Wild (1986) [2]. Also, Manski and Lerman (1977) show the same result in softmax setting [1].*

Unveiling the Significance: Translating this insight into the realm of machine learning, the constant (intercept) term is the bias term. This

fundamental observation reveals that when we recalibrate class weights or observation weights, the resulting changes primarily manifest as adjustments to the bias term. Put simply, the bias term acts as the linchpin connecting our strategy to address class imbalance.

A Unified Perspective

This understanding provides a straightforward explanation of how our algorithm, oversampling, and weight adjustment are, in essence, interchangeable and substitutable. This unification simplifies our approach while maintaining its potency in mitigating class imbalance challenges.

Simulation Study: Verifying the Bias Term Influence through Oversampling

To solidify our assertion that oversampling predominantly affects the bias term while keeping the model's functional core intact, we delve into a targeted simulation study. Our aim is to demonstrate empirically how oversampling techniques solely impact the bias term, leaving the model's essence unaltered.

The Simulation Setup

For this illustrative purpose, we focus on a simplified scenario: logistic regression with a single feature. Our model is defined as:

$$y_i = 1(2.5 + x_i + e_i > 0),$$

Created by Author

where 1(.) denotes the indicator function, x_i is drawn from a standard normal distribution, and e_i follows a logistic distribution. In this context, we set $f(x)=x$.

Running the Simulation:

Using this setup, we meticulously examine the impact of oversampling techniques on the bias term while keeping the model's core constant. We proceed with three oversampling methods: simple oversampling, SMOTE, and random sampling. Each method is meticulously applied, and the results are carefully recorded.

The Python code snippet below outlines the simulation process:

```
# Load Packages
import numpy as np
import statsmodels.api as sm
from imblearn.over_sampling import SMOTE, RandomOverSampler

# Set seed
np.random.seed(1)
# Create Simulation Datasets
x = np.random.normal(size = 10000)
y = (2.5 + x + np.random.logistic(size = 10000)) > 0
# Bias term is set to 2.5 and coefficient of x to 1

# The size of class 1 is 9005
print(sum(y == 1))
# The size of class 0 is 995
print(sum(y == 0))

# We want to match the size of class 0 to that of class 1
# Method 0 Don't do anything
x0 = x
y0 = y
method0 = sm.Logit(y0, sm.add_constant(x0)).fit()
print(method0.summary())  # 2.54 bias term and 0.97 x3 coefficient

# Method 1 Simple Oversampling
x1 = np.concatenate((x, np.repeat(x[y == 0], 8)))
```

```
y1 = np.concatenate((y, np.array([0] * (len(x1) - len(x)))))
method1 = sm.Logit(y1, sm.add_constant(x1)).fit()
print(method1.summary())  # 0.35 bias term and 0.98 x3 coefficient

# Method 2 SMOTE
smote = SMOTE(random_state = 1)
x2, y2 = smote.fit_resample(x[:, np.newaxis], y)
method2 = sm.Logit(y2, sm.add_constant(x2)).fit()
print(method2.summary())  # 0.35 bias term and 1 x3 coefficient

# Method 3 Random Sampling
random_sampler = RandomOverSampler(random_state=1)
x3, y3 = random_sampler.fit_resample(x[:, np.newaxis], y)
method3 = sm.Logit(y3, sm.add_constant(x3)).fit()
print(method3.summary()) # 0.35 bias term and 0.99 x3 coefficient
```

Results:

Method	Bias Term	Model Parameter
Nothing	2.54	0.97
Simple Oversampling	0.35	0.98
SMOTE	0.35	1
Random Oversampling	0.35	0.99

Simulation Results; Created by Author

Key Observations

The outcomes of our simulation study succinctly validate our proposition. Despite the application of various oversampling methods, the core model function $f(x)=x$ remains unaltered. The crucial insight lies in the remarkable consistency of the model component across all oversampling techniques. Instead, the bias term exhibits noticeable variations, corroborating our claim that oversampling primarily impacts the bias term without affecting the underlying model structure.

Reinforcing the Core Concept

Our simulation study undeniably underscores the fundamental equivalence between oversampling, weight adjustment, and bias term modification. By showcasing that oversampling exclusively alters the bias term, we fortify the principle that these strategies are interchangeable tools in the arsenal against class imbalance.

Applying the Bias Adjustment Algorithm to Credit Card Fraud Detection

To demonstrate the effectiveness of our bias adjustment algorithm in addressing class imbalance, we employ a real-world dataset from a Kaggle competition focused on credit card fraud detection. In this scenario, the challenge lies in predicting whether a credit card transaction is fraudulent (labeled as 1) or not (labeled as 0), given the inherent rarity of fraud cases.

We start by loading essential packages and preparing the dataset:

```
import numpy as np
import pandas as pd
import tensorflow as tf
import tensorflow_addons as tfa
from sklearn.model_selection import train_test_split
from imblearn.over_sampling import SMOTE, RandomOverSampler

# Load and preprocess the dataset
df = pd.read_csv("/kaggle/input/playground-series-s3e4/train.csv")
y, x = df.Class, df[df.columns[1:-1]]
x = (x - x.min()) / (x.max() - x.min())
x_train, x_valid, y_train, y_valid = train_test_split(x, y, test_size=0.3, rando
batch_size = 256
```

```
train_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train)).shuffle(b
valid_dataset = tf.data.Dataset.from_tensor_slices((x_valid, y_valid)).batch(bat
```

We then define a simple deep learning model for binary classification and set up the optimizer, loss function, and evaluation metric. I follow the competition evaluation and choose AUC as evaluation metric. Furthermore, the model is intentionally simplified as the focus of this article is to show how to implement the bias adjustment algorithm, not to ace in prediction:

```
model = tf.keras.Sequential([
    tf.keras.layers.Normalization(),
    tf.keras.layers.Dense(32, activation='swish'),
    tf.keras.layers.Dense(32, activation='swish'),
    tf.keras.layers.Dense(1)
])
optimizer = tf.keras.optimizers.Adam()
loss = tf.keras.losses.BinaryCrossentropy()
val_metric = tf.keras.metrics.AUC()
```

Within the core of our bias adjustment algorithm lies the training and validation steps, where we meticulously address class imbalance. To elucidate this process, we delve into the intricate mechanisms that balance the model's predictions.

Training Step with Accumulating Delta Values

In the training step, we embark on the journey of enhancing model sensitivity to class imbalance. Here, we calculate and accumulate the sum of model outputs for two distinct clusters: `delta0` and `delta1`. These clusters hold significant importance, representing the predicted values associated with classes 0 and 1, respectively.

```
# Define Training Step function
@tf.function
def train_step(x, y):
    delta0, delta1 = tf.constant(0, dtype = tf.float32), tf.constant(0, dtype =
    with tf.GradientTape() as tape:
        logits = model(x, training=True)
        y_pred = tf.keras.activations.sigmoid(logits)
        loss_value = loss(y, y_pred)
        # Calculate new bias term for addressing imbalance class
        if len(logits[y == 1]) == 0:
            delta0 -= (tf.reduce_sum(logits[y == 0]))
        elif len(logits[y == 0]) == 0:
            delta1 -= (tf.reduce_sum(logits[y == 1]))
        else:
            delta0 -= (tf.reduce_sum(logits[y == 0]))
            delta1 -= (tf.reduce_sum(logits[y == 1]))
    grads = tape.gradient(loss_value, model.trainable_weights)
    optimizer.apply_gradients(zip(grads, model.trainable_weights))
    return loss_value, delta0, delta1
```

Validation Step: Imbalance Resolution with Delta

The normalized delta values, derived from the training process, take center stage in the validation step. Armed with these refined indicators of class imbalance, we align the model's predictions more accurately with the true distribution of classes. The `test_step` function integrates these delta values to adaptively adjust predictions, ultimately leading to a refined evaluation.

```
@tf.function
def test_step(x, y, delta):
    logits = model(x, training=False)
    y_pred = tf.keras.activations.sigmoid(logits + delta)  # Adjust predictions
    val_metric.update_state(y, y_pred)
```

Utilizing Delta Values for Imbalance Correction

As training progresses, we collect valuable insights encapsulated within the `delta0` and `delta1` cluster sums. These cumulative values emerge as indicators of the bias inherent in our model's predictions. At the conclusion of each epoch, we execute a vital transformation. By dividing the accumulated cluster sums by the corresponding number of observations from each class, we derive normalized delta values. This normalization acts as a crucial equalizer, encapsulating the essence of our bias adjustment approach.

```
E = 1000
P = 10
B = len(train_dataset)
N_class0, N_class1 = sum(y_train == 0), sum(y_train == 1)
early_stopping_patience = 0
best_metric = 0
for epoch in range(E):
    # init delta
    delta0, delta1 = tf.constant(0, dtype = tf.float32), tf.constant(0, dtype =
    print("\nStart of epoch %d" % (epoch,))
    # Iterate over the batches of the dataset.
    for step, (x_batch_train, y_batch_train) in enumerate(train_dataset):
```



```
    # Take average of all delta values
    delta = (delta0/N_class0 + delta1/N_class1)/2

    # Run a validation loop at the end of each epoch.
    for x_batch_val, y_batch_val in valid_dataset:
        test_step(x_batch_val, y_batch_val, delta)

    val_auc = val_metric.result()
    val_metric.reset_states()
    print("Validation AUC: %.4f" % (float(val_auc),))
    if val_auc > best_metric:
        best_metric = val_auc
        early_stopping_patience = 0
    else:
        early_stopping_patience += 1
```

```
        if early_stopping_patience > P:
            print("Reach Early Stopping Patience. Training Finished at Validation AU
            break;
```

The Outcome

In our application to credit card fraud detection, the enhanced efficacy of our algorithm shines through. With bias adjustment seamlessly integrated into the training process, we achieve an impressive AUC score of 0.77. This starkly contrasts with the AUC score of 0.71 attained without the guiding hand of bias adjustment. The profound improvement in predictive performance stands as a testament to the algorithm's ability to navigate the intricacies of class imbalance, charting a course towards more accurate and reliable predictions.

References

[1] Manski, C. F., & Lerman, S. R. (1977). The estimation of choice probabilities from choice based samples. *Econometrica: Journal of the Econometric Society*, 1977–1988.

[2] Scott, A. J., & Wild, C. J. (1986). Fitting logistic models under case-control or choice based sampling. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, *48*(2), 170–182.

Imbalanced Data | Machine Learning | Data Science | Imbalanced Dataset | Imbalanced Class

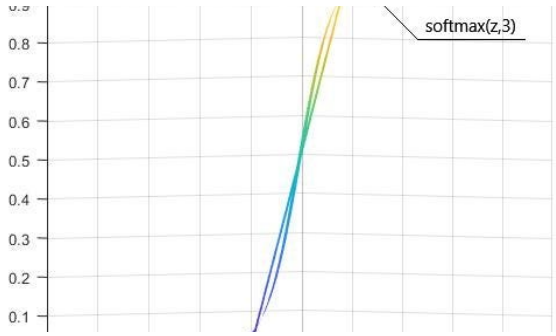
Written by Hyung Gyu Rho

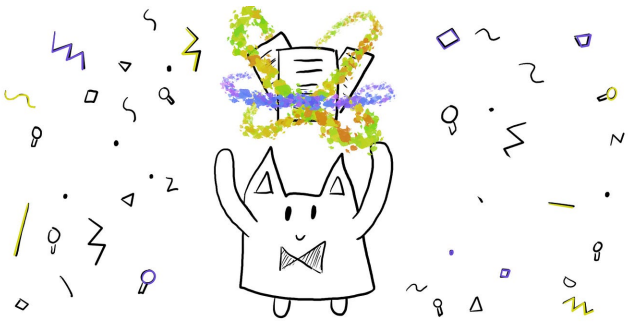
Following

16 Followers · Writer for Towards Data Science

I am an economics PhD at Penn State University. My main focus areas are econometrics and IO. Also, I am working on Machine Learning.

More from Hyung Gyu Rho and Towards Data Science





Hyung Gyu Rho in Towards Data Science

Improving The Inference Speed of TabNet

TabNet[1] is a deep neural network (dnn) based model for tabular data sets. The...

3 min read · Feb 25, 2021

Adrian H. Raudaschl in Towards Data Science

Forget RAG, the Future is RAG-Fusion

The Next Frontier of Search: Retrieval Augmented Generation meets Reciprocal...

· 10 min read · Oct 6

14

Antonis Makropoulos in Towards Data Science

How to Build a Multi-GPU System for Deep Learning in 2023

This story provides a guide on how to build a multi-GPU system for deep learning and…

10 min read · Sep 17

553 11

830 14

Mike Shakhomirov in Towards Data Science

How to Become a Data Engineer

A shortcut for beginners in 2024

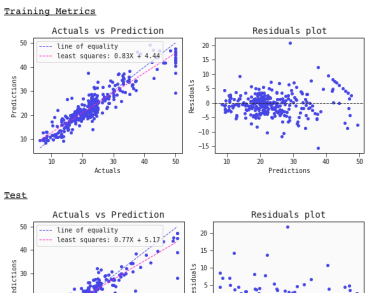
· 17 min read · Oct 7

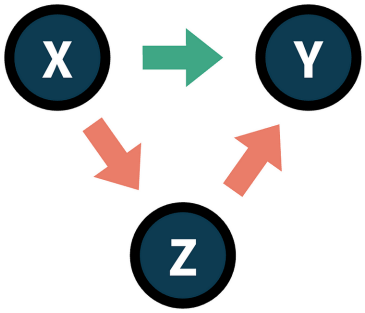
710 11

See all from Hyung Gyu Rho

See all from Towards Data Science

Recommended from Medium





Casper Skern Wilstrup

Symbolic Regression: a Simple and Friendly Introduction

Symbolic Regression is like a treasure hunt for the perfect mathematical equation to…

3 min read · May 5

82 1

Matteo Courthoud

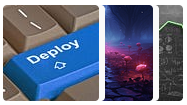
XYZ #5

A causal inference newsletter

2 min read · 5 days ago

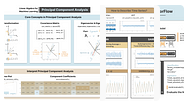
107

Lists



Predictive Modeling w/ Python

20 stories · 482 saves

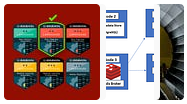


Practical Guides to Machine Learning

10 stories · 554 saves

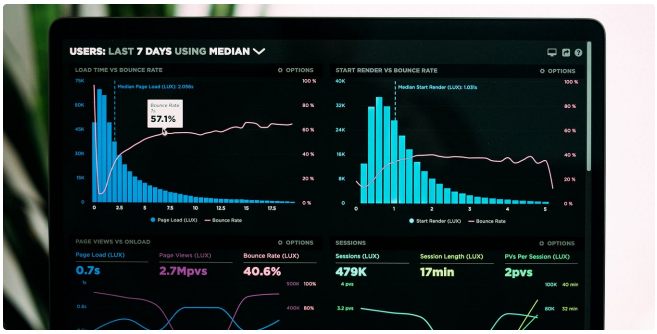
Natural Language Processing

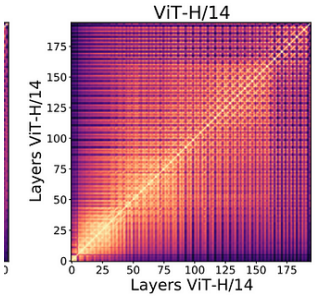
698 stories · 309 saves

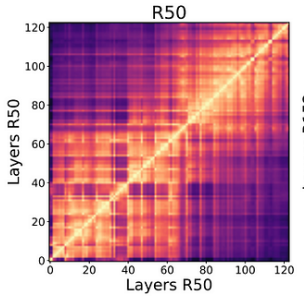


New_Reading_List

174 stories · 147 saves







Anton Yarkov in Better Programming

Algorithmic Alchemy: Exploiting Graph Theory in the Foreign…

If you're familiar with the FinTech startup industry, you may have heard of Revolut

15 min read · Oct 6

114

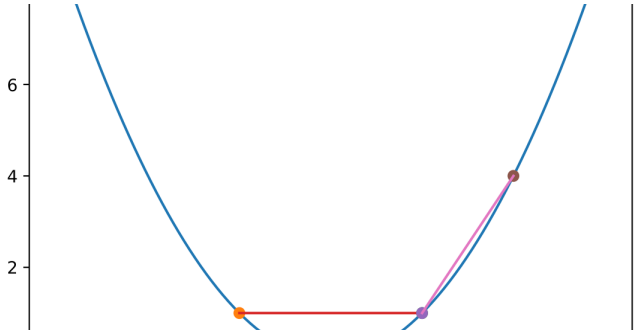
Ilias Papastratis

Comparison of Convolutional Neural Networks and Vision…

Introduction

19 min read · Sep 30

151 1



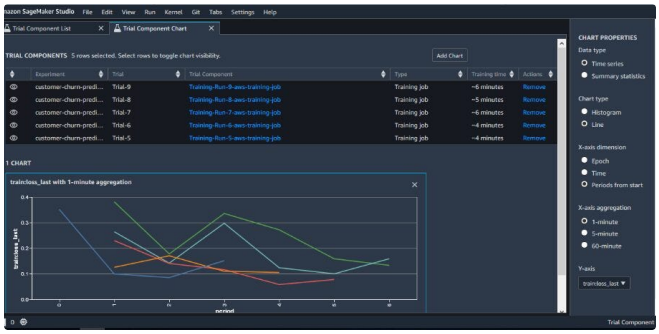
Saupin Guillaume in Level Up Coding

70 Mathematical Concepts: Linearization

Unveiling 70 Mathematical Concepts with Python

· 4 min read · 4 days ago

29 1



Yugank Aman

Top MLOps Tools to Manage Machine Learning Lifecycle

Businesses continue transforming their operations to increase productivity and…

10 min read · May 30

96

See more recommendations