令和６年度　特別研究報告書

顔認識とトラッキングを用いた傘管理システムの検討

龍谷大学　先端理工学部　知能情報メディア課程

Y210108 浮田郁久海

指導教員　三好　力　教授

内容梗概

近年、個人の所有物を効果的に管理し、紛失や盗難を防止する技術への関心が高まっている。本研究では、傘の盗難や誤持ち去りを防止するために、顔認識技術と物体トラッキング技術を組み合わせた傘管理システムを提案する。従来の鍵付き傘立てや GPS トラッキング付き傘には、利用者の手間や高コスト、GPS の追跡精度などの問題があり、実用性に限界があった。これに対し、本システムでは監視カメラを利用して利用者の顔をリアルタイムで認識し、傘の持ち主を特定する。また、物体検出技術により傘の位置をトラッキングし、持ち主以外の人物が傘に近づいた場合に警告を発する。このシステムは、既存の監視カメラを利用して導入コストを抑えることができる。これにより、商業施設や駅などの公共空間における傘の紛失防止を実現し、利用者の利便性を向上させることを目的としている。

目次

第 1 章　はじめに

1.1　研究背景

近年、個人の所有物を効果的に管理し、忘れ物、紛失や盗難を防止する技術への関心が高まっている。特に、公共の場で使用頻度が高い傘などの個人所有物は、意図せず他人に持ち去られるケースが頻発し、雨の日には多くの人が傘を他人のものと間違える、もしくは意図せずに盗難に遭うという問題が発生しており、解決策が求められている。図 1 に示す警視庁のホームページによる遺失物取扱状況では令和 5 年には東京都内だけで傘類は 294,196 点の傘が落とし物として届けられており、これは全体の 6.6%を占めており、特に駅や商業施設といった大勢の人が集まる場所で頻繁に起きている。

傘の忘れ物や紛失、盗難を防止するために、鍵付きの傘立てや GPS トラッカーを活用する方法が提案されているが、コストや手間が問題となり、広く普及しているとは言い難い状況にある。さらに、他人の傘を誤って持ち去るケースも多く、現代社会における課題の一つとなっている。そこで、本研究では顔認識と物体トラッキング技術を活用し、傘の所有者を自動的に識別し、他人による持ち去りや忘れ物を防ぐシステムの開発を目指します。

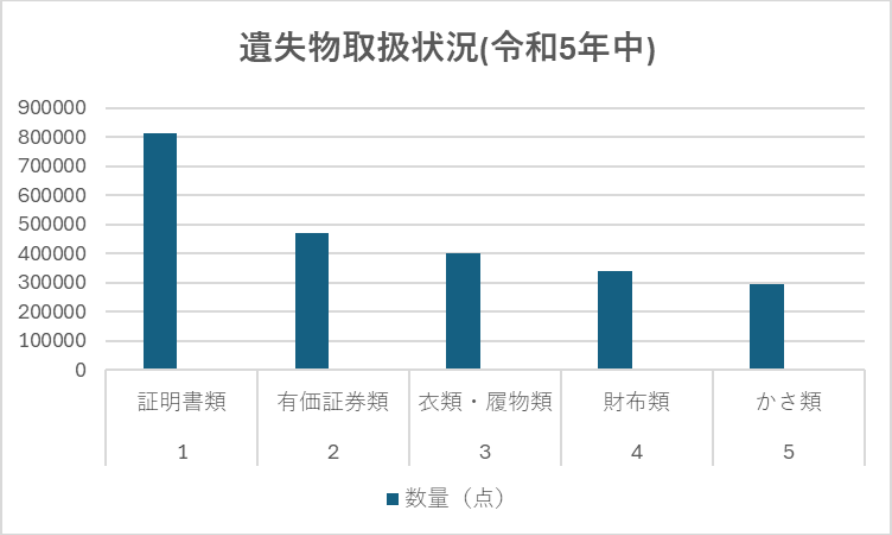

図 1　警視庁に届けられた遺失物

１．２　研究目的

本研究の目的は、既存の傘の盗難防止技術や管理システムが抱える課題を克服し、顔認識技術と物体トラッキング技術を組み合わせた新しい傘管理システムを提案することにある。従来の技術では、鍵の管理やコスト、GPS のバッテリーや精度など、いくつかの制約が存在していた。本研究では、それらの問題点を解決し、特に誤って他人の傘を持ち出してしまう状況を防ぐシステムの開発を目指す。最終的に、本システムを導入することで、低コストかつ効率的な傘管理ソリューションを提供し、商業施設や駅などでの傘の紛失や盗難、持ち去りといった問題の解決を目指す。またこのシステムは、監視カメラや既存のインフラを活用するため、特別な設備を必要とせず、公共の場での広範な適用が期待される。本システムが実用化されれば、商業施設や駅などの公共空間における傘の紛失防止を実現し、利用者の利便性を向上すると考えた。

１．３　既存技術

傘の盗難や紛失防止のため、いくつかの技術が導入されている。これらの技術はそれぞれ独自の方法で問題に対処しているが、いくつかの課題が存在する。以下に、代表的な技術とその問題点を述べる。

1.3.1　鍵付き傘立て

図２[*参考文献４]に示す鍵付き傘立ては、駅や商業施設でよく見られる盗難防止策であり、傘を物理的に固定して盗難を防止する仕組みである。利用者は傘を収納する際に専用の鍵を使い、他人が持ち出せないようにする。このシステムは盗難防止に一定の効果を発揮する一方で、利用者が鍵を紛失するリスクもあり、施錠を怠った場合には効果が発揮されないという問題がある。さらに、鍵付きの構造により製造コストが高くなり、鍵の故障や経年劣化による修理が必要になることもあり、維持費がかかる点がデメリットです。専用の施錠装置を設置するためにはコストが高額なため、全ての施設に導入するのは現実的ではない。

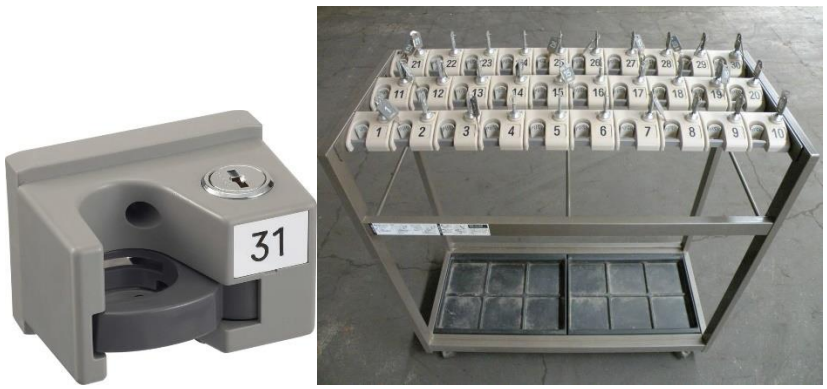

図２　鍵付き傘立ての例

1.3.2 GPS トラッキング付き傘や忘れ物トラッカー端末の活用

GPS トラッキング付き傘や後付けの忘れ物トラッカー端末（例：図 2 左の Apple 社が出しているトラッカー端末の AirTag や、図 3 右の MAMORIO 株式会社の製品である紛失防止デバイスの MAMORIO）[*参考文献 5]は、傘の位置を追跡することで紛失や盗難に対応する技術である。特に、GPS トラッキングは傘の位置をスマートフォンなどの端末で確認できるため、紛失時に見つけ出すことが容易になる。しかしながら、このシステムにも課題が存在する。まず、導入コストが高いことが挙げられ、日常的に使用される傘すべてに適用するのは難しい。さらに、GPS トラッカーには定期的なバッテリー交換が必要であり、利用者にとっての運用の煩雑さが問題となる。また、GPS を利用したものは屋内や地下では信号が不安定になり、追跡精度が低下するという制限がある。後付けのトラッカー端末に関しては、第三者によって簡単に取り外されてしまうリスクがあり、盗難時には機能しない場合もある。盗難や紛失防止に一定の効果がある一方で、利用者の手間やコスト、技術的な制約があるため、実用的で広く普及する解決策とは言いがたい。

図 3　紛失防止デバイスの例

第 2 章　提案手法

第 1 章で述べた問題点を解決する策として、利用者が鍵を紛失するリスクを無くし、低コストかつ効率的な傘管理ソリューションを検討した。本研究では、顔認識技術と物体検出・トラッキング技術を組み合わせ、傘の持ち主をリアルタイムで特定し、他人がその傘に手にした際に警告を発するシステムを提案する。

まず、傘の所有者がシステムに顔を登録し、その後カメラを通じて所有者をリアルタイムで識別する。これにより、持ち主の傘と他人の傘を区別することができる。本研究では物体検出のためのアルゴリズムである YOLO を使用して、カメラの映像から傘を検出・追跡し、持ち主と傘の位置をリアルタイムでトラッキングし、他人が傘に近づいた場合や手に取った場合に警告を発する。この機能により、持ち主が意図せず他人の傘を持ち去ることや他人が誤って持ち主の傘を取るのを防ぐ。これにより、他人が傘に近づいた瞬間にユーザーに警告を出すことで盗難防止が期待できる。図 4 に本システムのフローチャートを示す。

GPS トラッカーや施錠システムに比べ、顔認識カメラと物体トラッキング技術を活用するため、既存のカメラ設備を活かしたシステムの導入が可能です。また、広い範囲での導入が容易であり、公共施設や商業施設でも利用が期待される。

この提案手法は、既存の施錠型傘立てや GPS トラッキング付き傘のように物理的な制約に頼らず、顔認識と物体トラッキングを組み合わせることで、より実用的でより低コストな解決策を提供する。また、傘の持ち主がカメラのフレーム内にいる限り、持ち主の傘を守り続けるという機能を備えているため、特に大勢が集まる公共の場所での活用が期待される。

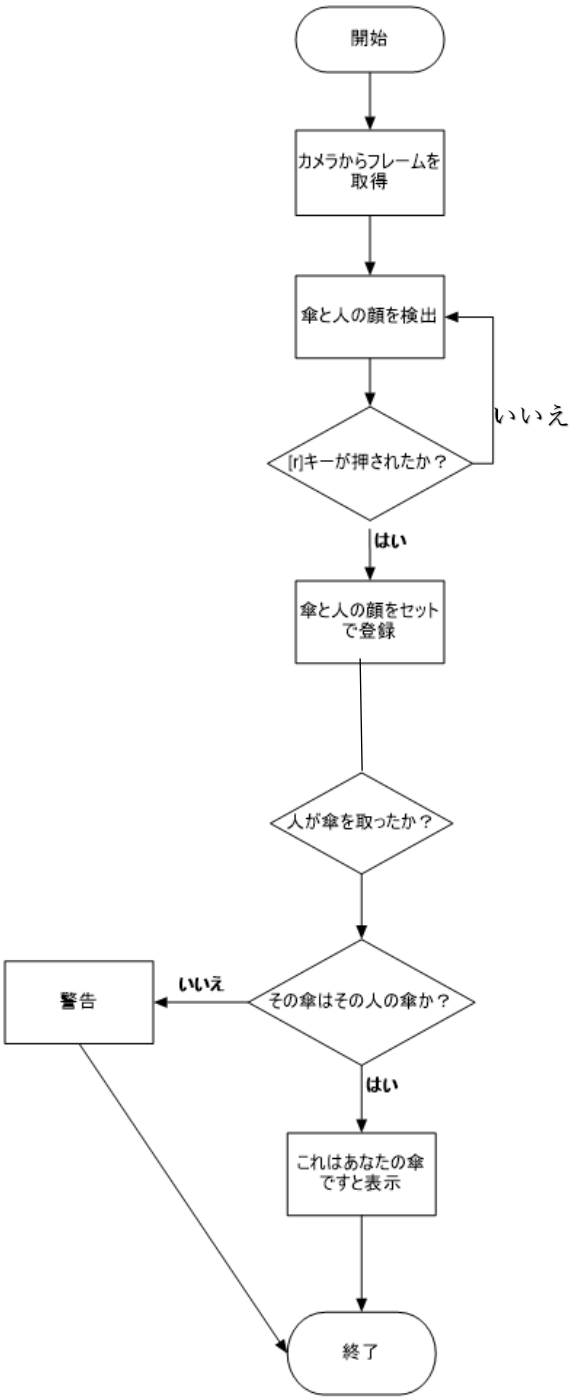

図４　本システムのフローチャート

3章　実験手法

3.1 顔認識

顔認識は、カメラ映像から人の顔を検出し、その顔を特定の特徴に基づいて識別することを目的とする。本研究では、Python の face_recognition ライブラリ[6]を用いて顔検出と顔認識を行う。このライブラリは、検出された顔の位置が示され、プログラムがどのように顔を認識しているかを視覚的に確認できるようにしている。

3.1.1 顔の特徴量の抽出

顔の特徴量の抽出は、顔認識システムにおいて重要な工程の一つである。顔の特徴量とは、各顔の個別の特徴を数値化したものであり、これにより顔の固有のパターンを数値的に表現できる。本研究では、各顔の特徴量を 128 次元のベクトルとして抽出し、このベクトルはそれぞれの顔に固有のものである。128 次元のベクトルは、顔の形状や輪郭、目、鼻、口などの特徴を詳細に捉え、数値化することで、機械的に顔の一致や識別が可能となる。この特徴量は顔認識のために保存し、後で他の顔と照合する際に用いることで、プログラムは同一人物の顔を識別しやすくなり、精度の高い認識を実現する。

3.1.2 顔の登録

顔の登録はシステムが新たな顔を認識し、後の照合に使用するために行うものである。本研究では、R キーを押すことで新しい顔が登録され、顔の特徴量が予め用意されたリストに追加される仕組みを採用している。このリストには、登録された顔ごとに ID が付与され、各顔と ID の紐付けが行われる。これにより、登録された顔は個別に管理され、複数の顔が登録されても ID をもとに識別が可能となる。顔が新たに登録されることで、プログラムは次回以降、登録済みの顔と照合を行うことができ、登録済みの人物が映像内に現れた際に即座に認識される。顔の登録機能によって、システムは複数の人物を同時に管理し、識別できるように設計している。

提案するシステムでは、登録処理を自動化したいと考えているが、実験では顔を登録するために、R キーを押すことで登録されるようにしている。提案する手法としてはプライバシーを考慮して顔の自動登録後、傘がなくなった時点で自動消去されるように顔の特徴量をリストから削除することを考えている。

3.1.3 顔の照合

顔の照合機能は、登録済みの顔とカメラ映像内の顔との一致を判断するために重要な役割を果たすものである。登録済みの顔がある場合、検出された顔と登録された顔の特徴量を比較し、類似度を計算することで顔の一致を判断する。本研究では、検出された顔の特徴量と登録された顔の特徴量の間の距離が 0.3 以下である場合を「一致している」と見なす。この値が小さいほど、顔が似ている、つまり同一人物であると判断できる。0.3 という閾値を採用することで、誤認識の可能性を低減しつつ、精度の高い顔照合を実現している。

さらに、照合の結果、顔が一致している場合は該当する顔 ID が画面上に表示され、カメラ映像上に人物の情報が視覚的に示される。一方、一致しない場合には、その顔のラベルには「Unknown」と表示され、登録されていない顔であることが示されるため、ユーザーは登録されていない人物であることを即座に認識できる。この「Unknown」の表示は、システムに登録されていない人物が映像内にいることを視覚的に把握できる。

顔照合機能の導入によって、セキュリティシステムや監視システムなど、多くの分野で顔認識システムの活用が期待される。

3.2 傘の検出

傘の検出には、YOLO（You Only Look Once）モデル[7]を使用する。YOLO は、物体検出に特化したディープラーニングモデルで、画像全体を一度に処理することで物体のクラスとその位置（バウンディングボックス）を同時に推定し検出する特徴を持つ。この技術により、リアルタイムでの物体検出が可能となり、システムの応答速度が大幅に向上する。本研究においては、傘の検出を効率よく行うために、YOLOv8 を採用し、さらに傘の検出精度を向上させるために、独自にデータセットを作成して学習モデルをファインチューニングした。これにより、YOLOv8 が傘の位置を迅速かつ高精度に検出できるようにしている。

3.3 データセットの作成

傘の検出精度を向上させるためには、良質なデータセットが不可欠である。そのため、インターネット上の公開データセットに加え、実際の使用環境における傘の画像を手動で撮影し、データを収集した。このデータ収集の際、傘の種類、形状、サイズ、色、使用状況など、幅広いシチュエーションでのデータが揃うように工夫している。例えば、屋内での傘の保管状態や屋外での傘の使用状況など、多様なシーンで傘がどのように映り込むかを考慮してデータ収集を行った。これにより、異なる環境での傘の検出精度が向上し、一般的な傘以外にも透明なビニール傘や折りたたみ傘などの特殊な傘も正確に検出できるようにしている。収集したデータは、訓練用と検証用のデータセットに分け、モデルの学習と性能評価に使用した。

3.3.1 アノテーションの実施

収集した画像データに対して、傘の位置と輪郭を正確に指定するため、オープンソースのアノテーションツール「LabelImg」[8]を使用してアノテーションを行った。LabelImg は、画像内の物体に対して矩形のバウンディングボックスを描画し、その範囲を指定してラベルを付けることができるツールである。このアノテーション作業では、傘が映る角度や形状、色、使用状況など、可能な限り多様な状態の傘をラベル付けするように心がけた。これにより、YOLO モデルが訓練される際に、幅広いシチュエーションでの傘の位置を高精度に検出できるようになる。また、アノテーション作業の際には、傘と他の物体（例えば人や椅子）との位置関係も考慮し、誤検出が発生しないように工夫し、傘検出の信頼性を高めている。

3.3.2 YOLO モデルの再学習

YOLO モデルは、事前学習済みモデルとして一般的な物体検出用に広範なデータで訓練されているが、傘に特化した検出を行うためには、再学習（ファインチューニング）が必要である。本研究では、YOLOv8 の事前学習済みモデルをベースに、収集した傘専用のデータセットを用いてファインチューニングを実施

した。この際、エポック数は 500、バッチサイズは 16 に設定し、モデルが安定した結果を出せるようにしている。500 エポックという学習回数は、モデルが過学習を避けつつも、十分に傘の特徴を学習できる回数である。また、バッチサイズを 16 とすることで、GPU メモリの制限を考慮しつつ効率的に学習を進められるようにした。ファインチューニング後、傘の検出精度が向上し、複雑な背景や他の物体と重なった場合でも正確に傘を検出できるようになった。この再学習によって、YOLOv8 モデルが本研究の用途に最適化され、リアルタイムでの傘検出が可能となった。

3.4 傘の特徴量の保存

傘の特徴量を保存するために本システムでは、3 つの特徴量を取得し、傘の照合に使用している。本システムで使用する特徴量は LBP（Local Binary Pattern、ORB（Oriented FAST and Rotated BRIEF)、色ヒストグラムの各特徴量の類似度に重み付けを行い、それを基に総合スコアを計算する。

3.4.1 LBP（Local Binary Pattern）特徴量

LBP（Local Binary Pattern）は、画像のテクスチャを特徴付ける手法であり、主に物体の表面パターンや模様を解析するために利用される。この手法は、各ピクセルの周囲に配置されたピクセルの輝度値とその中心ピクセルの輝度値を比較し、その相対的な差からバイナリパターンを生成することに基づき、計算される。具体的には、周囲のピクセル値が中心ピクセル値よりも大きければ 1、小さければ 0 と判定し、これを時計回りまたは反時計回りに並べることで 2 進数パターンを構築します。この 2 進数パターンを 10 進数に変換した値を LBP 値と呼びます。LBP 値は、画像の小領域ごとに計算され、それらを集計してヒストグラムを作成します。このヒストグラムがその領域のテクスチャを表す特徴量となります。

LBP の特徴は、照明条件の変化や輝度の違いに対して非常に頑健である点です。そのため、明るさやコントラストが異なる画像であっても同様のパターンを検出できるという強みがあります。また、計算が高速であるため、リアルタイム処理にも適しており、顔認識や物体認識などの分野で広く使用されています。しかしながら、スケールや回転に対する頑健性は限定的であり、細かな模様や複雑な色彩変化には対応しにくいという課題もあります。

傘の特徴量解析においては、LBP は特に傘表面の生地や模様のテクスチャを抽出するのに有効だと考えて採用した。単色の傘や布地の違いを判別するのに適しており、他の特徴量と組み合わせることで、識別性能をさらに向上させることができると考える。

3.4.2 ORB（Oriented FAST and Rotated BRIEF）特徴

ORB（Oriented FAST and Rotated BRIEF）は、画像の特徴点を検出し、それを記述する効率的なアルゴリズムであり、物体認識や画像マッチングの分野で広く利用されている。ORB は、FAST（Features from Accelerated Segment Test）と BRIEF（Binary Robust Independent Elementary Features）という 2 つのアルゴリズムを基に設計されており、これらの欠点を克服するための改良が加えられている。特にスケール不変性と回転不変性を備えている点が特徴であり、様々なアプリケーションでの利用が可能である。

ORB では、まず FAST アルゴリズムを使用して特徴点を検出する。FAST は、画像中の角やエッジのような特徴のある領域を迅速に検出する手法であるが、単体では特徴点の重要度を評価する仕組みがない。このため、ORB では Harris コーナー検出に基づくスコアリングが追加されており、信頼性の高い特徴点を選別することができるようになっている。また、ORB ではスケール不変性を持たせるため、画像のピラミッド構造を生成し、異なる解像度で特徴点を検出している。この手法により、画像のサイズが異なっていても同じ特徴点を検出することが可能である。

さらに、回転不変性を実現するために、各特徴点の周囲の勾配方向を計算し、それを基準として特徴点を回転させる処理が行われる。これにより、画像が回転していても特徴点を正確に検出し、比較することが可能となっている。この特性により、ORB はスケールや回転の影響を受けにくい頑健なアルゴリズムであるといえる。

特徴点の記述には BRIEF アルゴリズムが用いられている。BRIEF は、特徴点周辺のピクセル間の輝度値を比較し、その結果をバイナリ（0 または 1）のビット列として記述する手法である。しかし、BRIEF は回転不変性を持たないという欠点があるため、ORB では特徴点の主方向に基づいて記述子を回転させる改良が加えられている。この改良により、ORB は回転に対する頑健性を備えた特徴記述子を生成することができる。

ORB 記述子の比較にはハミング距離が用いられている。これは、バイナリビット列間で異なるビット数を計算することで類似度を測定する手法であり、計算コストが低く、リアルタイム処理にも適している。ORB の記述子を用いることで、画像中の物体の特徴点を効率的に比較し、一致度を評価することが可能である。

傘の認識システムにおいては、ORB は傘の形状や模様を特徴付けるために重要な役割を果たしている。登録された傘の ORB 記述子と、フレーム内で検出された傘の ORB 記述子を比較することで、傘の一致度を判定し、所有権を確認することができる。また、ORB は計算が高速でありながら高い識別性能を持つため、リアルタイムの傘認識システムにおいて非常に有効である。

これにより、傘のように比較的単純な形状の物体でもその特徴を正確に捉え、物体認識や追跡システムの一部として効果的に機能するアルゴリズムであるといえる。

3.4.3 色ヒストグラム特徴

色ヒストグラムは、画像内の色の分布を特徴付けるための手法であり、特定の領域における各色の出現頻度を数値化したものである。この手法は、物体の色の特徴を捉える際に非常に有効であり、主に画像認識や物体追跡、画像検索などの分野で使用されている。

色ヒストグラムの計算では、画像を RGB や HSV といった色空間に分解し、それぞれのチャンネルごとに一定の範囲を持つビン（区間）を設定して、その範囲に含まれるピクセル数を数える。例えば、RGB の場合、それぞれのチャンネルに 8 つのビンを設定した場合、全体で 8×8×8=512 通りの色分布が表現可能となる。このようにして得られるヒストグラムは、画像全体または特定の領域の色の特徴を表す数値ベクトルとなる。

正規化することで、ヒストグラムが画像のスケール（ピクセル数）や照明条件に依存しない形に調整され、正規化されたヒストグラムは、他の画像との比較や類似度の計算に適している。

傘の認識システムにおいて色ヒストグラムは、特に傘の模様が少なく、単色または色のパターンが特徴的である場合に効果的である。登録時に計算された傘の色ヒストグラムと、現在フレーム内で検出された傘のヒストグラムを比較することで、その一致度を測定し、傘の所有権を判定する基準となる。たとえば、コサイン類似度を用いて登録済みの傘と現在の傘の色分布が高い一致率を示す場合、その傘は登録されたものと判断することができる。

色ヒストグラムの強みは、計算が簡単で、形状や位置に依存せずに物体を識別できる点である。また、照明条件が大きく変化しない環境であれば、非常に安定した性能を発揮する。一方で、色が似通った異なる物体を区別するのは難しく、模様が複雑である場合や光の影響で色が変化する場合には、精度が低下する可能性がある。

色ヒストグラムは、傘のように単純な形状でありながらカラフルな特徴を持つ物体の識別において効果的であると判断したため採用した。

3.5 傘のトラッキング

傘の検出後、傘のトラッキングを行う。YOLO によってフレームごとに傘が検出されると、それぞれの傘に固有の ID を割り当て、複数フレームにわたって傘の位置情報を追跡する。このトラッキング機能により、フレーム内で傘が移動しても、同一の傘として認識され続けることが可能になる。トラッキングには、各フレームで傘の位置をリストに保存し、ID ごとに管理する方法を採用している。これにより、システムは複数の傘が同時に映像内に登場しても、個別に追跡できる。

さらに、トラッキング情報を蓄積することで、傘の移動パターンを分析し、傘の所有者がどのように傘を使用しているかを把握することも可能となる。このトラッキング技術は、複数の傘が混在する環境において、各傘を適切に識別し続けるために非常に重要な役割を果たしている。トラッキング結果は、リアルタイムで画面に表示され、傘の位置が視覚的に確認できるようになっている。

3.5.1 YOLO の特性とトラッキングにおける課題の克服

YOLO（You Only Look Once）は、高速かつ高精度な物体検出を実現するディープラーニングベースのアルゴリズムであり、リアルタイム処理が求められるアプリケーションで広く使用されている。その大きな特徴は、物体検出を単一のニューラルネットワークの推論で完了させるという設計にある。従来の物体検出手法に比べて大幅な速度向上を実現しつつ、精度においても競争力が高い。これにより、監視カメラ映像やロボット制御、ドライバーアシスタンスシステムなど、さまざまな応用分野で利用が進んでいる。

しかしながら、YOLO の設計は主に各フレーム内での物体検出に最適化されており、フレーム間の連続性や物体の追跡に関する機能を持たない。そのため、YOLO 単体では、物体トラッキングに必要なフレーム間の情報の結び付けを効果的に処理できない。特に、検出結果の位置揺れや物体の一時的な消失への対応が困難であり、同じ物体を継続的に追跡する際には、これらの課題が大きな制約となる。

このような課題を解決するためには、カルマンフィルターや SORT (Simple Online and Realtime Tracking) といった手法を組み合わせることが有効である。これらの技術は、YOLO が提供する検出結果を利用し、フレーム間での物体トラッキングを安定化させる。具体的には、カルマンフィルターと SORT はそれぞれ異なる役割を果たしつつ、相互に補完的な効果をもたらす。

SORT は、物体トラッキングに特化したアルゴリズムであり、YOLO の検出結果を基に物体の位置を予測し、それを観測値と組み合わせて更新する手法を取る。このプロセスにより、物体が一時的に視界から消えた場合でも、予測値を基にトラッキングを継続できる。また、YOLO の検出結果には一定のノイズが含まれる場合があるが、SORT は観測値を用いて予測値を補正することで、ノイズを低減し滑らかな位置情報を提供する。さらに、SORT はフレーム間で同一の物体に対して一貫したトラッキング ID を付与する機能

を持つ。このトラッキング ID は、特に複数の物体を同時に追跡する場合に重要であり、同じ物体をフレーム間で正確に識別するために不可欠である。

具体的に SORT が提供する機能には、物体の位置予測、観測値の更新、トラッキング ID の維持が含まれる。物体の位置予測は、現在の位置と速度を基に次のフレームでの位置を推定するものであり、予測アルゴリズムによって物体が一時的に隠れた場合にもトラッキングが途切れないようにする。また、観測値の更新では、YOLO が提供する検出結果を観測値として取り込み、予測値と統合して精度を向上させる。このプロセスにより、検出結果のばらつきや不安定性を抑え、滑らかな物体の位置情報が得られる。さらに、トラッキング ID の維持によって、フレーム間で同じ物体に一貫した識別子を付与し、誤認識を防止することが可能である。

加えて、カルマンフィルターは、トラッキング精度をさらに向上させるために有用な手法である。カルマンフィルターは、「予測」と「更新」という 2 つのステップを繰り返すことで、高精度な位置情報を提供するアルゴリズムである。まず、過去の位置情報と速度を基に次のフレームでの位置を予測する。次に、YOLO の検出結果を観測値として取り込み、予測値を補正する。このプロセスにより、YOLO の検出結果に含まれる位置揺れを抑えつつ、正確な位置情報を得ることができる。さらに、カルマンフィルターの大きな利点は、検出が一時的に途切れた場合でも、予測値に基づいてトラッキングを継続できる点にある。これにより、一時的に物体が視界から消えたとしても、トラッキングの一貫性が損なわれない。

カルマンフィルターと SORT を組み合わせることで、YOLO の物体検出性能を最大限に活用しながら、トラッキングの安定性と精度を大幅に向上させることができる

3.6 傘と持ち主の結び付け

本システムでは、ユーザーが「R」キーを押すことで、顔 ID と傘の特徴量が結び付けられる仕組みを実装している。登録時に、システムはまずカメラ映像内の顔を検出し、その顔に ID を付与する。次に、顔の位置に最も近い傘を検索し、顔 ID と傘を関連付ける。具体的には、顔のバウンディングボックス中心と傘のバウンディングボックス中心の距離を計算し、その距離が最小となる傘を選択することで、顔 ID と傘を 1 対 1 で結び付ける。この方法により、誤認識を避けつつ、同一人物が持っている傘と顔が正確に対応付けられる。

加えて、顔のエンコーディング（特徴量）はリストに保存され、その ID は別のリストに登録される。一方、傘に関しては、傘の一意の識別 ID および傘の特徴量を抽出し、これらの情報を辞書に格納する。これによりシステム内で顔と傘のペアが登録され、リアルタイムで追跡と認識が行えるようにしており、後の照合にも役立てられる。

4 章　結果

4.1　顔認証の精度

前章 3.1 の顔認証の登録が正常に動作したかの確認を行う。未登録の顔には unknown と表示され、登録された顔には、通し番号で ID が振り分けられていく。未登録の顔が検出されたときの例を図 5 に示す。登録済みの顔が検出されたときの例を図 6 に示す。顔認証の精度については登録時と同じような角度であれば 80%~90%の精度で本人と認識できている。その結果を図 8 に示す。

角度がずれている場合でも７０％以上の信頼度があり、顔認証は成功していると言える。

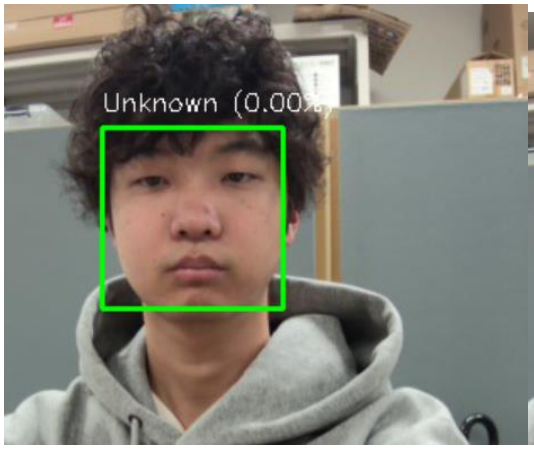

図 5　未登録の顔の検出時の例

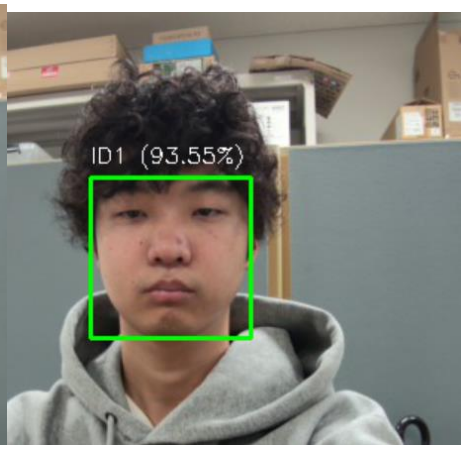

図 6　登録済みの顔の検出時の例

4.2 傘の検出の精度

前章 3.2 の傘の検出が正常に動作したかの確認を行う。傘の検出には、YOLOv8 の事前学習済みモデルを基盤として、収集した傘専用のデータセットを用い、ファインチューニングを実施した。これにより、傘を高精度に検出できるようになっている。図 7 には、検出結果の一例を示している。図中では、検出された傘が矩形で囲まれており、その検出信頼度も併せて表示されている。その結果を図 9 に示す。

現時点での検出精度はおおよそ 75%前後であり、この結果は本研究の目的において実用的な精度と評価できる。ただし、モデル学習時のパラメータをさらに最適化することで、検出精度の向上が期待できる

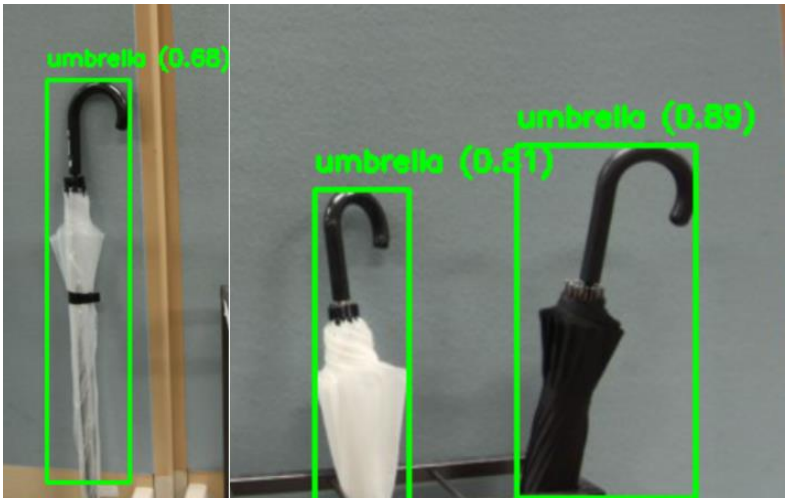

図 7　傘検出時の例

4.3 前節 4.1 4.2 の精度の結果

前節 4.1 4.2 の精度の確認を３０回ずつ行った。それぞれの結果を図 8、図 9 に示す。

実験回数	顔の認識率
1	0.826655918
2	0.83560516
3	0.860046658
4	0.888762603
5	0.882839595
6	0.823444774
7	0.8219829
8	0.889993071
9	0.850977799
10	0.880538563
11	0.875439291
12	0.897576373
13	0.83917519
14	0.816202597
15	0.874776109
16	0.808188692
17	0.747654227
18	0.728446108
19	0.771271878
20	0.786639783
21	0.814346142
22	0.818262693
23	0.816329563
24	0.821016447
25	0.793293516
26	0.7866896
27	0.79404051
28	0.823503876
29	0.809370544
30	0.81961238
平均	**0.826756085**

図 8　顔認証の精度の結果

実験回数	傘の認識率
1	0.841613531
2	0.747781157
3	0.805744827
4	0.813081384
5	0.796570718
6	0.752257288
7	0.753157139
8	0.803227544
9	0.731375754
10	0.83645159
11	0.710759223
12	0.801351904
13	0.803227544
14	0.841732235
15	0.732181549
16	0.813099265
17	0.834503174
18	0.846432209
19	0.827522755
20	0.756967604
21	0.740393043
22	0.704316854
23	0.774018943
24	0.834897041
25	0.833191276
26	0.832348406
27	0.845168829
28	0.855706989
29	0.82217735
30	0.746735573
平均	**0.794599757**

図 9　傘の検出の精度

4.4 傘の持ち主判定

前章 3.6 の傘と持ち主の結び付けが正常に動作したかの確認を行う。人が傘を取り、その傘がその人物が登録した傘であれば傘の上に緑の文字で「Your umbrella 」と表示させる。その時の例を図 10 に示す。その傘がその人物が登録した傘でなければ赤の文字で「NOT Your umbrella 」と表示させ、その時の例を図 11 に示す。

図 10　所有者の傘の場合の例

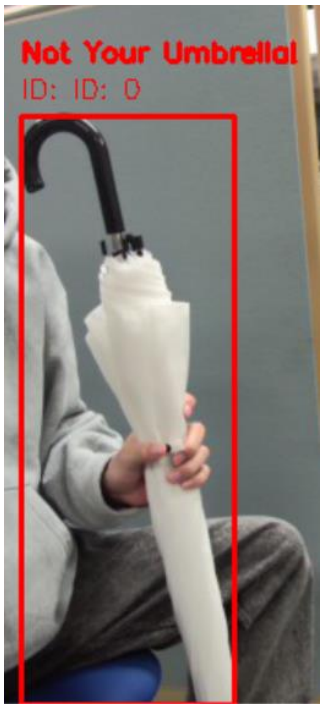

図 11　所有者ではない傘の場合

4.5　システムの動作実験の結果

人が傘を登録し、正しい傘を取った場合と、間違った傘を取った場合の各特徴量取得するという実験を３０回ずつ行った。それぞれの結果を図１２、図１３に示す。

正解			
実験回数	ORB	SSIM	ヒストグラム
1	0.67	0.43	0.99
2	0.58	0.43	0.99
3	0.5	0.42	0.99
4	0.67	0.43	0.99
5	0.58	0.43	0.99
6	0.67	0.43	0.99
7	0.13	0.32	0.96
8	0.27	0.42	0.98
9	0.5	0.36	0.98
10	0.6	0.42	0.99
11	0.33	0.42	0.97
12	0.5	0.42	0.97
13	0.42	0.42	0.98
14	0.66	0.43	0.86
15	0.44	0.45	0.87
16	0.38	0.47	0.87
17	0.33	0.47	0.86
18	0.33	0.43	0.87
19	0.33	0.47	0.88
20	0.47	0.47	0.86
21	0.47	0.47	0.87
22	0.43	0.47	0.87
23	0.44	0.47	0.99
24	0.38	0.42	0.88
25	0.44	0.42	0.86
26	0.42	0.43	0.86
27	0.38	0.42	0.87
28	0.33	0.43	0.87
29	0.44	0.47	0.99
30	0.33	0.44	0.85
平均	0.44733	0.432667	0.925

図 12　正しい傘を取った場合の表

不正解			
実験回数	ORB	SSIM	ヒストグラム
1	0.13	0.38	0.57
2	0.1	0.24	0.5
3	0.11	0.22	0.51
4	0.07	0.25	0.59
5	0.03	0.27	0.57
6	0.05	0.22	0.55
7	0.12	0.22	0.57
8	0.16	0.28	0.5
9	0.17	0.25	0.59
10	0.01	0.25	0.57
11	0.02	0.25	0.52
12	0.07	0.31	0.53
13	0.13	0.31	0.54
14	0.09	0.31	0.54
15	0.13	0.26	0.54
16	0.05	0.33	0.54
17	0.08	0.26	0.53
18	0.12	0.27	0.53
19	0.11	0.3	0.53
20	0.8	0.22	0.53
21	0.07	0.37	0.58
22	0.09	0.34	0.58
23	0.03	0.35	0.5
24	0.06	0.28	0.58
25	0.1	0.28	0.52
26	0.09	0.3	0.59
27	0.17	0.26	0.5
28	0.18	0.26	0.55
29	0.09	0.31	0.56
30	0.07	0.28	0.53
平均	0.116667	0.281	0.544666667

図 13　間違った傘を取った場合の表

正しい傘を取った場合の ORB の特徴量は平均して、約 0.45 であり、間違った傘を取った場合の ORB の特徴量の平均は約 0.12 であった。

SSIM の正しい傘を取った場合の平均値は約 0.43 であり、間違った傘を取った場合の平均値は約 0.28 であった。

ヒストグラムの正しい傘を取った場合の平均値は約 0.92 であり、間違った傘を取った場合の平均値は約 0.54 であった。

いずれの特徴量も正しい傘を取った場合と間違った傘を取った場合で値が大きく減少することを確認できた。以上のことから特徴量の平均値が「正しい傘」と「間違った傘」で大きく異なることから、このシステムは実際に傘の識別が可能であると考えられる。

本研究では、ビニール傘 2 本、白色の傘、黒色の傘、灰色の傘を使用した。ビニール傘の判別は課題が残るものの、その他の傘の判別には成功したと言える。しかし、本システムでは、トラッキングを用いて傘の ID を保持しているため素早い動きや傘が重なりすぎていると、トラッキングが失敗し判別が困難になるという課題がある。

5 章　考察

本章では、本実験の結果の考察、および本システムの評価と今後の課題と展望について述べる

5.1 実験結果に対する考察

本研究では、AI 技術を活用した傘識別システムを構築し、その有効性を確認した。ORB 特徴量、SSIM、ヒストグラム類似度という 3 種類の特徴量を用いることで、「正しい傘」と「間違った傘」の識別が可能であることが示された。実験結果から、正しい傘を取った場合と間違った傘を取った場合で、それぞれの特徴量において明確な差異が確認され、システムの識別力の高さを実証できた。特に、ヒストグラム類似度は正しい傘で平均 0.92、間違った傘で平均 0.54 と大きな差を示し、最も信頼性の高い識別基準であることが示唆されたことから傘の取り間違いや忘れ物、盗難を防止に有効であると考えた

5.2　課題

本システムは傘の識別に一定の成功を収めたが、以下の課題が明らかになった。

まず、ビニール傘の識別精度が不十分である点が挙げられる。ビニール傘は形状や色、テクスチャが非常に似通っており、ORB 特徴量、SSIM、ヒストグラム類似度といった基本的な特徴量では十分な差異を抽出できず、誤検出が多発する傾向が見られた。このため、深層学習を用いた高度な特徴抽出や、テクスチャ解析の強化が必要である。

次に、トラッキング性能の限界が課題となった。特に、傘の動きが速い場合や複数の傘が近接・重なった場面では、トラッキングが失敗し、ID の追跡が途切れるケースが確認された。これにより、正確な識別が困難になる場面が生じた。DeepSORT や ByteTrack のような外見特徴を活用した高度なトラッキングアルゴリズムを導入することで、これらの問題に対応できる可能性があり、改善の余地がある。

さらに、実運用環境への対応も課題である。照明条件の変化や、傘が部分的に隠れる遮蔽状態では、検出精度が低下する可能性がある。本研究では実装できなかったが複数カメラの導入やカメラ配置の最適化を検討する必要がある。

以上の課題を解決することで、本システムの実用性がさらに向上し、さまざまな運用環境での信頼性が確保できると考えられる。

6. まとめ

本研究では傘の盗難や持ち去りを防止するため顔認識技術と物体トラッキング技術を組み合わせた傘管理システムを提案し、評価実験を行った。

実験では各特徴量も正しい傘を取った場合と間違った傘を取った場合で値が大きく減少することを確認でき、傘の判別は可能という結果が得られたが、ビニール傘のような見た目が酷似している傘の識別には精度が良くないことが課題として明らかになった。

今後の展望としては、課題として挙げたビニール傘の識別精度向上やトラッキング性能の強化に加え、改良することでより広範な利用シーンで活用されることが期待できると考えている。

謝辞

本論文を作成するにあたり、多くのご指導、ご助言をいただきました三好 力教授に深く感謝の意を表します。

参考文献

[１]自作データセットの作成

YOLOv8 自作データセット https://qiita.com/zakutakumi/items/008c77cd8d9c7c3cd00b

[２] RFID による忘れ物防止システムの実現性の考察

https://ipsj.ixsq.nii.ac.jp/ej/?action=pages_view_main&active_action=repository_view_main_item_detail&item_id=105683&item_no=1&page_id=13&block_id=8

[３] 気付きを誘起する忘れ物防止支援システム　https://uec.repo.nii.ac.jp/records/5014

[４]図１錠付き傘立ての例の写真　https://x.gd/DtvMl

[5] 図２　紛失防止デバイスの例 https://www.apple.com/jp/airtag/　　https://x.gd/fBxJj

[6]face_recognition ライブラリ https://pystyle.info/perform-face-detection-with-python/

[7] YOLO(You Only Look Once) https://docs.ultralytics.com/ja/models/yolov8/

[8]アノテーションツール「LabelImg」https://laid-back-scientist.com/labelimg

付録　ソースコード

```
import cv2
import face_recognition
import numpy as np
from ultralytics import YOLO
from collections import defaultdict, deque
from filterpy.kalman import KalmanFilter
import torch
from skimage.feature import local_binary_pattern
from sklearn.metrics.pairwise import cosine_similarity

# 傘の持ち主の登録情報
registered_faces = []
face_labels = []
umbrella_registry = {}  # {人物 ID: (傘のボックス, トラッキング ID, 特徴情報, 平均色)}
track_history = defaultdict(lambda: deque(maxlen=10))  # 傘 ID ごとのトラッキング履歴（最新 10 個）

# ORB 特徴量検出器
orb = cv2.ORB_create()
device = torch.device("cpu")  # CPU のみに限定
model = YOLO(r"C:¥Users¥ikumi¥runs¥segment¥train8¥weights¥best.pt").to(device)
print(f"Using device: {device}")

# カメラ設定
cap = cv2.VideoCapture(0)
cap.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)
w, h, fps = (int(cap.get(x)) for x in (cv2.CAP_PROP_FRAME_WIDTH, cv2.CAP_PROP_FRAME_HEIGHT, cv2.CAP_PROP_FPS))
out = cv2.VideoWriter("umbrella-tracking-final.avi", cv2.VideoWriter_fourcc(*"MJPG"), fps, (w, h))

frame_count = 0  # フレームカウンタ
next_label = 1   # ユーザーID 用のカウンタ

# --- KalmanFilter を使用したトラッカー ---
class KalmanBoxTracker:
    count = 0

    def __init__(self, bbox):
        """Kalman Filter 初期化"""
        self.kf = KalmanFilter(dim_x=7, dim_z=4)
        self.kf.F = np.eye(7)
        self.kf.F[4:, 4:] += 1  # 速度を考慮
        self.kf.H = np.eye(4, 7)
        self.kf.P[4:, 4:] *= 1000.0
        self.kf.P *= 10.0
        self.kf.R *= 1.0
        self.kf.Q[-1, -1] *= 0.01
        self.kf.Q[4:, 4:] *= 0.01
        self.kf.x[:4] = bbox.reshape((4, 1))
        self.time_since_update = 0
        self.id = KalmanBoxTracker.count
        KalmanBoxTracker.count += 1

    def update(self, bbox):
        self.kf.update(bbox)

    def predict(self):
        self.kf.predict()
        self.time_since_update += 1
        return self.kf.x[:4].reshape((1, 4))

    def get_state(self):
        return self.kf.x[:4].reshape((1, 4))

# --- SORT アルゴリズムを使用した複数物体トラッキング ---
class Sort:
    def __init__(self, max_age=1, min_hits=3, iou_threshold=0.3):
        self.trackers = []
        self.max_age = max_age
        self.min_hits = min_hits
        self.iou_threshold = iou_threshold

    def update(self, detections):
        """トラッカーの更新"""
        for tracker in self.trackers:
            tracker.predict()

        # 新しいトラッカーの作成
        for i in range(len(detections)):
            self.trackers.append(KalmanBoxTracker(detections[i, :]))

        # 古いトラッカーの削除
        self.trackers = [t for t in self.trackers if t.time_since_update <= self.max_age]

        return np.array([t.get_state() for t in self.trackers if t.time_since_update == 0])

sort_tracker = Sort()

def calculate_average_color(image):
    """
    画像の平均色を計算して返す。
    """
    return np.array(cv2.mean(image)[:3])

def calculate_color_similarity(color1, color2):
    """
    2 つの色の類似度（0〜1）を計算。
    color1, color2: RGB 値（3 次元ベクトル）
    """
    distance = np.linalg.norm(color1 - color2)
    max_distance = np.sqrt(255**2 * 3)
    return 1 - (distance / max_distance)

def check_umbrella_ownership(face_encodings, face_locations, tracked_objects, im0):
    """
    傘の特徴量（色や ID）を基に、映っている人の傘かどうかを判定。
    """
    for face_encoding, face_location in zip(face_encodings, face_locations):
        distances = face_recognition.face_distance(registered_faces, face_encoding)
        if len(distances) > 0 and np.min(distances) < 0.4:
            matched_face_id = face_labels[np.argmin(distances)]
            top, right, bottom, left = face_location

            # 人物を枠で囲む
            cv2.rectangle(im0, (left, top), (right, bottom), (0, 255, 0), 2)
                    cv2.putText(im0, f"Person {matched_face_id}", (left, top - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)

            for obj in tracked_objects:
                x1, y1, x2, y2 = map(int, obj[0])  # 傘の位置
                umbrella_image = im0[y1:y2, x1:x2]
                umbrella_color = calculate_average_color(umbrella_image)

                # 傘が登録済みかを判定
                if matched_face_id in umbrella_registry:
                                registered_box, registered_track_id, _, registered_color = umbrella_registry[matched_face_id]

                    # 色の類似度を計算
                                color_similarity = calculate_color_similarity(umbrella_color, registered_color)

                    # ID が一致しているか確認
                    is_same_id = tracked_objects.index(obj) == registered_track_id

                    if color_similarity > 0.8 or is_same_id:  # 類似度が高いか ID が一致
                        text = "Your Umbrella"
                        color = (0, 255, 0)  # 緑
                    else:
                        text = "Not Your Umbrella"
                        color = (0, 0, 255)  # 赤

                    # 傘を枠で囲む
                    cv2.rectangle(im0, (x1, y1), (x2, y2), color, 2)
                    cv2.putText(im0, text, (x1, y1 - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, color, 2)

def calculate_lbp(image, P=8, R=1):
    """
    LBP 特徴量を計算。
    P: 隣接点の数
    R: 半径
    """
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    lbp = local_binary_pattern(gray, P, R, method="uniform")
    hist, _ = np.histogram(lbp.ravel(), bins=np.arange(0, P + 3), range=(0, P + 2))
    hist = hist.astype("float")
    hist /= hist.sum()  # 正規化
    return hist

def calculate_orb_features(image):
    """
    ORB 特徴量を計算。
    """
    orb = cv2.ORB_create()
    keypoints, descriptors = orb.detectAndCompute(image, None)
    return descriptors

def calculate_histogram(image, bins=(8, 8, 8)):
    """
    色ヒストグラムを計算。
    bins: 各色チャンネルのビン数
    """
    hist = cv2.calcHist([image], [0, 1, 2], None, bins, [0, 256, 0, 256, 0, 256])
    hist = cv2.normalize(hist, hist).flatten()
    return hist

def calculate_histogram_similarity(hist1, hist2):
    """
    ヒストグラム間の類似度を計算（コサイン類似度）。
    hist1, hist2: ヒストグラム（正規化済みの配列）
    """
    return cosine_similarity([hist1], [hist2])[0][0]

# ---  傘の矩形と ID を描画 ---
def display_umbrella_id(tracked_objects, im0):
    """
    傘の矩形を登録状況に応じて描画。
    登録済みの傘は青色の矩形で囲み、ID を常に表示。
    """
    for idx, obj in enumerate(tracked_objects):
        x1, y1, x2, y2 = map(int, obj[0])  # 傘の位置情報
        umbrella_id = f"ID: {idx}"

        # 傘が登録済みかどうかをトラッキング ID で判定
        is_registered = False
                for _, (registered_box, registered_track_id, registered_features) in umbrella_registry.items():
            # 登録されたトラッキング ID と現在の ID を比較
            if registered_track_id == idx:
                is_registered = True
                break

        # 矩形とテキストの描画
        if is_registered:
            color = (255, 0, 0)  # 青色
            text = f"Registered: {umbrella_id}"
        else:
                color = (0, 0, 255)  # 赤色
                text = f"Unregistered: {umbrella_id}"

        # 描画処理
```

```
        cv2.rectangle(im0, (x1, y1), (x2, y2), color, 2)
        cv2.putText(im0, text, (x1, y1 - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, color, 1)

# --- IOU（Intersection over Union）計算 ---
def calculate_iou(box1, box2):
    """
    2 つの矩形の IOU（Intersection over Union）を計算する。
    box1, box2 は[x1, y1, x2, y2]形式のリストまたはタプル。
    """
    x1 = max(box1[0], box2[0])
    y1 = max(box1[1], box2[1])
    x2 = min(box1[2], box2[2])
    y2 = min(box1[3], box2[3])

    # 重なり領域の面積（Intersection）
    inter_width = max(0, x2 - x1)
    inter_height = max(0, y2 - y1)
    inter_area = inter_width * inter_height

    # 各矩形の面積
    box1_area = (box1[2] - box1[0]) * (box1[3] - box1[1])
    box2_area = (box2[2] - box2[0]) * (box2[3] - box2[1])

    # 結合領域の面積（Union）
    union_area = box1_area + box2_area - inter_area

    # IOU 計算
    if union_area == 0:
        return 0  # Union が 0 になる場合は IOU も 0
    iou = inter_area / union_area
    return iou

def check_left_side_umbrella_color(face_encodings, face_locations, tracked_objects, im0):
    """
    登録された傘のヒストグラム特徴量と現在左側で検出されている傘のヒストグラムを比較。
    違う場合は「NOT Your Umbrella」と警告を表示。
    """
    frame_width = im0.shape[1]
    left_side_threshold = frame_width // 2

    for face_encoding, face_location in zip(face_encodings, face_locations):
        distances = face_recognition.face_distance(registered_faces, face_encoding)
        if len(distances) > 0 and np.min(distances) < 0.4:
            matched_face_id = face_labels[np.argmin(distances)]
            top, right, bottom, left = face_location

            # 登録された傘のヒストグラムを取得
            if matched_face_id in umbrella_registry:
                registered_box, registered_track_id, registered_features = umbrella_registry[matched_face_id]
                registered_histogram = registered_features['hist']  # 登録されたヒストグラム

                for obj in tracked_objects:
                    x1, y1, x2, y2 = map(int, obj[0])  # 傘の位置

                    # 傘が左側にある場合
                    if x2 < left_side_threshold:
                        umbrella_image = im0[y1:y2, x1:x2]
                        detected_histogram = calculate_histogram(umbrella_image)  # 現在のヒストグラムを計算

                        # ヒストグラム間の類似度を計算
                        histogram_similarity = calculate_histogram_similarity(detected_histogram, registered_histogram)

                        if histogram_similarity < 0.8:  # 類似度が低い場合
                            text = "NOT Your Umbrella"
                            color = (0, 0, 255)  # 赤色
                        else:  # 類似度が高い場合
                            text = "Your Umbrella"
                            color = (0, 255, 0)  # 緑色

                        # 傘を枠で囲む
                        cv2.rectangle(im0, (x1, y1), (x2, y2), color, 2)
                        cv2.putText(im0, text, (x1, y1 - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, color, 2)
                        print(f"比較結果: 類似度 {histogram_similarity:.2f}, {text}")

def calculate_similarity(registered_features, detected_features):
    """
    特徴量の類似度を計算して総合スコアを返す。
    """
    # 特徴量が存在しない場合の処理
    if registered_features is None or detected_features is None:
        print("警告: 特徴量が不足しています。類似度計算をスキップします。")
        return 0, 0, 0, 0

    # LBP 類似度（コサイン類似度）
    try:
        lbp_similarity = cosine_similarity(
            [registered_features['lbp']], [detected_features['lbp']]
        )[0][0]
    except Exception as e:
        print(f"エラー: LBP 類似度の計算に失敗しました: {e}")
        lbp_similarity = 0

    # ORB 類似度（特徴点の一致率）
    try:
        bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
        matches = bf.match(registered_features['orb'], detected_features['orb'])
        orb_similarity = len(matches) / max(len(registered_features['orb']), len(detected_features['orb']))
    except Exception as e:
        print(f"エラー: ORB 類似度の計算に失敗しました: {e}")
        orb_similarity = 0

    # ヒストグラム類似度（コサイン類似度）
    try:
        hist_similarity = cosine_similarity(
            [registered_features['hist']], [detected_features['hist']]
        )[0][0]
    except Exception as e:
        print(f"エラー: ヒストグラム類似度の計算に失敗しました: {e}")
        hist_similarity = 0

    # 総合スコア（重み付け）
    total_score = 0.4 * lbp_similarity + 0.3 * orb_similarity + 0.3 * hist_similarity
    return total_score, lbp_similarity, orb_similarity, hist_similarity

# --- 傘と持ち主を登録 ---
def register_person_and_umbrella(face_encoding, face_id, umbrella_box, umbrella_track_id, umbrella_image):
    """
    傘と人物の顔情報を登録。
    """
    registered_faces.append(face_encoding)
    face_labels.append(face_id)

    # 特徴量を計算して登録
    features = {
        'lbp': calculate_lbp(umbrella_image),
        'orb': calculate_orb_features(umbrella_image),
        'hist': calculate_histogram(umbrella_image)
    }

    # 特徴量が計算できていない場合の対処
    if features['lbp'] is None or features['orb'] is None or features['hist'] is None:
        print(f"警告: 傘の特徴量が正しく計算できませんでした。登録をスキップします。")
        return
    umbrella_registry[face_id] = (umbrella_box, umbrella_track_id, features)
    print(f"登録完了: 顔 ID: {face_id}, 傘 ID: {umbrella_track_id}")
# --- 傘の所有者を確認 ---
def check_umbrella_ownership(face_encodings, face_locations, tracked_objects, im0):
    """
    傘の特徴量を基に、現在検出されている傘が登録された傘かを判定。
    """
    for face_encoding, face_location in zip(face_encodings, face_locations):
        distances = face_recognition.face_distance(registered_faces, face_encoding)
        top, right, bottom, left = face_location

        if len(distances) > 0 and np.min(distances) < 0.4:
            matched_face_id = face_labels[np.argmin(distances)]
            top, right, bottom, left = face_location

            # 登録された傘の特徴量を取得
            if matched_face_id in umbrella_registry:
                registered_data = umbrella_registry[matched_face_id]
                registered_box, registered_track_id, registered_features = registered_data[:3]  # 最初の 3 つを取得

                for obj in tracked_objects:
                    x1, y1, x2, y2 = map(int, obj[0])  # 傘の位置
                    umbrella_image = im0[y1:y2, x1:x2]

                    # 現在検出された傘の特徴量を計算
                    detected_features = {
                        'lbp': calculate_lbp(umbrella_image),
                        'orb': calculate_orb_features(umbrella_image),
                        'hist': calculate_histogram(umbrella_image)
                    }

                    # 特徴量を比較
                    total_score, lbp_similarity, orb_similarity, hist_similarity = calculate_similarity(
                        registered_features, detected_features
                    )

                    # 判定結果を表示
                    if total_score > 0.7:  # スコアが一定以上なら本人の傘
                        text = "Your Umbrella"
                        color = (0, 255, 0)  # 緑色
                    else:
                        text = "NOT Your Umbrella"
                        color = (0, 0, 255)  # 赤色
                    # 傘を枠で囲む
                    cv2.rectangle(im0, (x1, y1), (x2, y2), color, 2)
                    cv2.putText(im0, text, (x1, y1 - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, color, 2)
                    print(f"スコア: {total_score:.2f}（LBP: {lbp_similarity:.2f}, ORB: {orb_similarity:.2f}, Hist: {hist_similarity:.2f}）")
        else:
            cv2.rectangle(im0, (left, top), (right, bottom), (0, 0, 255), 2)
            cv2.putText(im0, "Unknown", (left, top - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 255), 1)
# --- フレーム処理 ---
def process_frame(im0):
    global frame_count, next_label
    frame_count += 1
    frame_width = im0.shape[1]  # フレームの幅を取得
    left_side_threshold = frame_width // 2
    results = model.predict(im0)
    detections = results[0].boxes.xyxy.cpu().numpy()

    tracked_objects = sort_tracker.update(detections)

    # 傘の ID を描画
    display_umbrella_id(tracked_objects, im0)

    # 顔の検出と登録
    face_locations = face_recognition.face_locations(im0, model="hog")
    face_encodings = face_recognition.face_encodings(im0, face_locations)

    if cv2.waitKey(1) & 0xFF == ord("r"):  # "r"キーで登録
        if len(face_encodings) > 0 and len(detections) > 0:
            face_encoding = face_encodings[0]  # 最初の顔を使用
            top, right, bottom, left = face_locations[0]
            x1, y1, x2, y2 = map(int, detections[0])  # 最初の傘を取得
            umbrella_box = detections[0]

            # 傘と顔がフレームの左半分にある場合のみ登録
            if  x2 < left_side_threshold:
```

```
                    register_person_and_umbrella(face_encoding, next_label, umbrella_box,
next_label, im0[y1:y2, x1:x2])
                    next_label += 1
                    print(f"登録完了: 傘と顔を検出（傘 ID: {next_label - 1}）")
            else:
                print("登録失敗: 傘または顔が左半分にありません。")

    # 傘の所有者確認
    check_umbrella_ownership(face_encodings, face_locations, tracked_objects, im0)
    check_left_side_umbrella_color(face_encodings, face_locations, tracked_objects, im0)
# --- メインループ ---
while True:
    ret, im0 = cap.read()
    if not ret:
        print("フレームの取得に失敗しました。")
        break
    process_frame(im0)
    out.write(im0)
    cv2.imshow("umbrella-tracking", im0)

    if cv2.waitKey(1) & 0xFF == ord("q"):
        break

out.release()
cap.release()
cv2.destroyAllWindows()
```