Aldina Kurtović

Razvoj softvera 2

10.01.2026.

Implementacija recommender sistema

TaxiMo je mobilna aplikacija za naručivanje taksi usluga koja povezuje putnike i vozače na osnovu njihove dostupnosti, lokacije i prethodnih iskustava korisnika. Kako aplikacija raste i broj registrovanih vozača postaje sve veći, korisnicima može biti teško da uvijek dobiju vozača koji najbolje odgovara njihovim preferencijama (npr. kvalitet vožnje, cijena, trajanje vožnje ili vrijeme dana).

Zbog toga je u aplikaciji implementiran **recommender sistem za preporuku vozača**, čiji je cilj da korisniku ponudi **najprikladnije vozače** na osnovu njegovih prethodnih vožnji i recenzija, čime se poboljšava korisničko iskustvo i povećava zadovoljstvo putnika.

4. Preporuka vozača (Driver Recommendation System)

Kako bi korisniku bio ponuđen najprikladniji vozač, sistem koristi **personalizovani preporučivački sistem zasnovan na mašinskom učenju**, implementiran pomoću **ML.NET biblioteke** i **content-based filtering pristupa**.
Model uči isključivo na osnovu **historije vožnji i recenzija konkretnog korisnika**, bez oslanjanja na podatke drugih korisnika.

4.1 Učitavanje i provjera modela

Prilikom svakog zahtjeva za preporuku vozača, sistem provjerava da li postoji već istrenirani model za datog korisnika:

- Ako model postoji – učitava se i koristi odmah za predikciju.
- Ako model ne postoji (npr. novi korisnik ili invalidiran model) – sistem automatski trenira novi model na osnovu dostupnih podataka o prethodnim vožnjama korisnika.

Modeli su perzistentno spremljeni u obliku **.zip datoteka**, čime se izbjegava ponovno treniranje pri svakom zahtjevu i poboljšavaju performanse sistema.

4.2 Priprema podataka

Model se trenira isključivo na osnovu **završenih vožnji** za koje postoji korisnička recenzija.

Svaka vožnja se transformiše u skup numeričkih karakteristika koje opisuju vozača i samu vožnju.

Za treniranje se koriste sljedeće karakteristike:

- **Prosječna ocjena vozača** – indikator kvaliteta i reputacije vozača
- **Ukupan broj završenih vožnji vozača** – indikator iskustva
- **Prosječna cijena vožnje** – modelira korisnikovu osjetljivost na cijenu
- **Udaljenost vožnje (km)**
- **Trajanje vožnje (minute)**
- **Dio dana** (jutro, popodne, noć) – hvata vremenske preferencije korisnika

Kao ciljna vrijednost (label) koristi se **kontinuirani skor zadovoljstva korisnika**, izveden iz ocjene vožnje:

- 1.0 – visoko zadovoljstvo (4–5 zvjezdica)
- 0.6 – neutralno iskustvo (3 zvjezdice)
- 0.2 – nezadovoljstvo (1–2 zvjezdice)

Ovakav pristup omogućava modelu da uči **nijansirane preferencije korisnika**, umjesto binarne podjele na dobre i loše vozače.

4.3 Treniranje i čuvanje modela

Za treniranje se koristi **SDCA regresijski algoritam (Stochastic Dual Coordinate Ascent)**, koji je pogodan za regresione probleme sa većim brojem numeričkih karakteristika.

Model se:

- trenira samo kada je potrebno (lazy training),
- sprema u datoteku vezanu za korisnika,
- ponovo trenira isključivo kada se pojave novi relevantni podaci.

Model se automatski **invalidira** nakon:

- završetka vožnje,
- dodavanja nove recenzije.

Na taj način sistem osigurava da preporuke uvijek reflektuju **najnovije ponašanje korisnika**, bez nepotrebnog opterećenja sistema.

4.4 Predikcija i rangiranje vozača

Prilikom generisanja preporuka, sistem:

1. Identifikuje trenutno dostupne (slobodne) vozače.
2. Za svakog vozača formira ulazni vektor karakteristika.
3. Koristi istrenirani ML model za izračunavanje **predikcionog skora**.
4. Sortira vozače po predikcionom skoru (opadajuće).
5. Vraća **Top N preporučenih vozača** (default: 5).

Predikcioni skor predstavlja vjerovatnoću da će korisnik biti zadovoljan vozačem, na osnovu njegovih prethodnih iskustava.

5. Lične preferencije korisnika

Lične preferencije korisnika u sistemu TaxiMo **ne modeliraju se eksplicitno kao poseban heuristički modul**, već se **implicitno uče kroz mašinski model** na osnovu historije prethodnih vožnji i recenzija korisnika.

Sistem analizira sljedeće obrasce ponašanja korisnika:

- **Tip vozača koje korisnik preferira**, indirektno kroz:
  - prosječnu ocjenu vozača koje je korisnik pozitivno ocijenio,
  - iskustvo vozača (broj završenih vožnji).
- **Cjenovne preferencije korisnika**, na osnovu:
  - prosječne cijene vožnji koje je korisnik koristio i visoko ocijenio.
- **Preferencije u vezi trajanja i dužine vožnje**, kroz:
  - udaljenost vožnje (km),
  - trajanje vožnje (minute).
- **Vremenske preferencije**, kroz:
  - dio dana u kojem korisnik najčešće koristi uslugu (jutro, popodne, noć).

Na osnovu ovih podataka, model uči **lični preferencijski profil korisnika**, te generiše **predikcioni skor** koji odražava koliko se konkretni vozač uklapa u dosadašnje obrasce ponašanja korisnika.

Drugim riječima, lični score nije ručno izračunat, već predstavlja **rezultat naučenih preferencija korisnika**, enkodiranih unutar regresijskog modela.

---

6. Filtriranje i prikaz preporuka

Prije konačnog prikaza preporučenih vozača, sistem primjenjuje skup poslovnih i sigurnosnih pravila.

U konačni izbor ulaze samo vozači koji:

- su trenutno **dostupni (slobodni)** za prihvatanje vožnje,
- imaju validne i potpune podatke potrebne za generisanje preporuke,
- su rangirani na osnovu predikcionog skora dobijenog ML modelom.

Sistem zatim:

1. Sortira vozače po predikcionom skoru (opadajuće).
2. Ograničava rezultat na **Top-N preporuka** (zadana vrijednost: 5).
3. Prikazuje preporučene vozače korisniku unutar mobilne aplikacije, u posebnoj sekciji *Recommended Drivers*.

Ovakav pristup osigurava da korisnik vidi **relevantne i personalizovane preporuke**, uz istovremeno zadržavanje jednostavnog i preglednog korisničkog interfejsa.

7. Cold Start – novi korisnici

U slučaju kada korisnik nema dovoljno historijskih podataka (npr. novi korisnik bez završenih vožnji ili recenzija), sistem ne može pouzdano trenirati personalizovani model.

U tom slučaju se primjenjuje **cold start strategija**, pri čemu sistem:

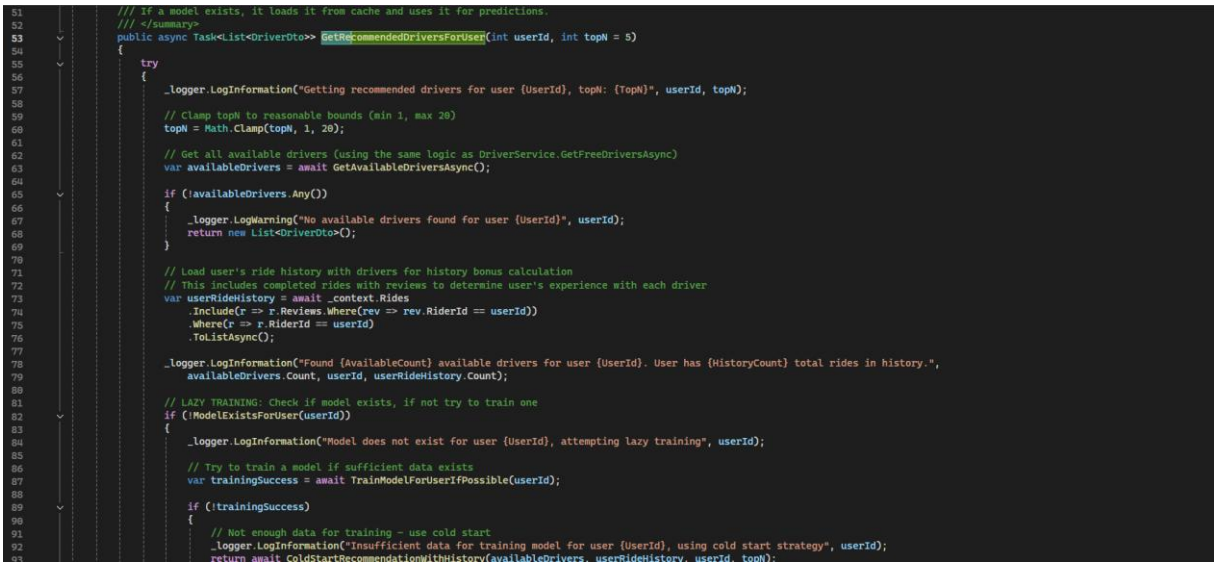
- koristi osnovne informacije o vozačima,
- preporučuje vozače sa:
    - većim brojem završenih vožnji,
    - višom prosječnom ocjenom.

Na ovaj način novi korisnici i dalje dobijaju **razumne i kvalitetne preporuke**, dok sistem postepeno prikuplja podatke potrebne za izgradnju personalizovanog modela.

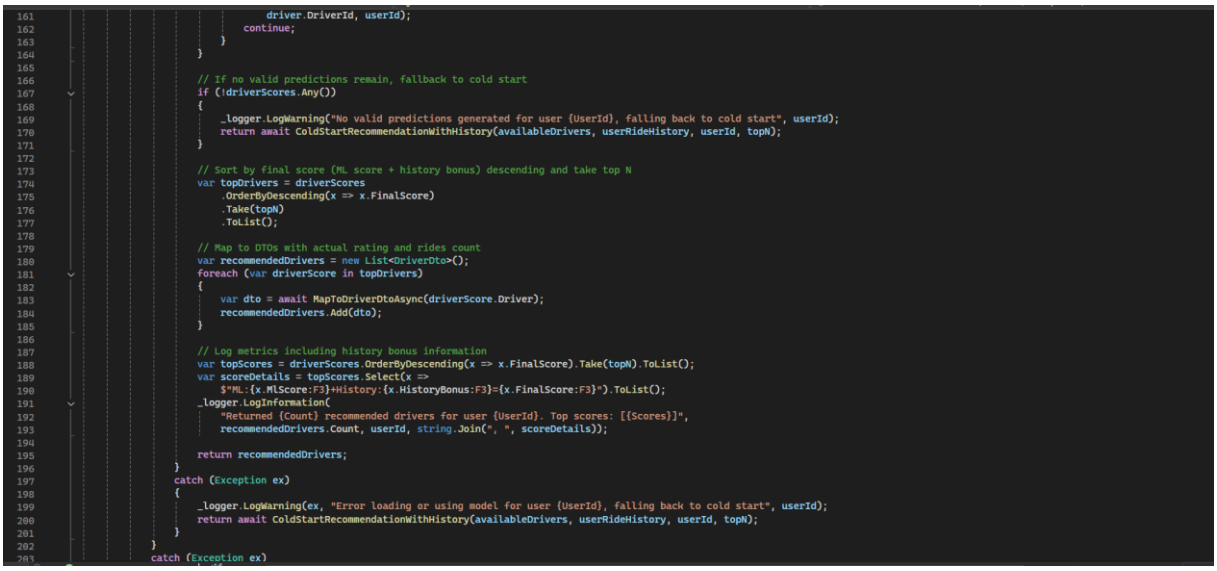
Kako korisnik počne završavati vožnje i ostavljati recenzije, sistem automatski prelazi sa cold start pristupa na **potpuno personalizovane preporuke**.

Putanja do source code-a:
TaxiMo\TaxiMo\TaxiMo.Services\Services\DriverRecommendationService.cs

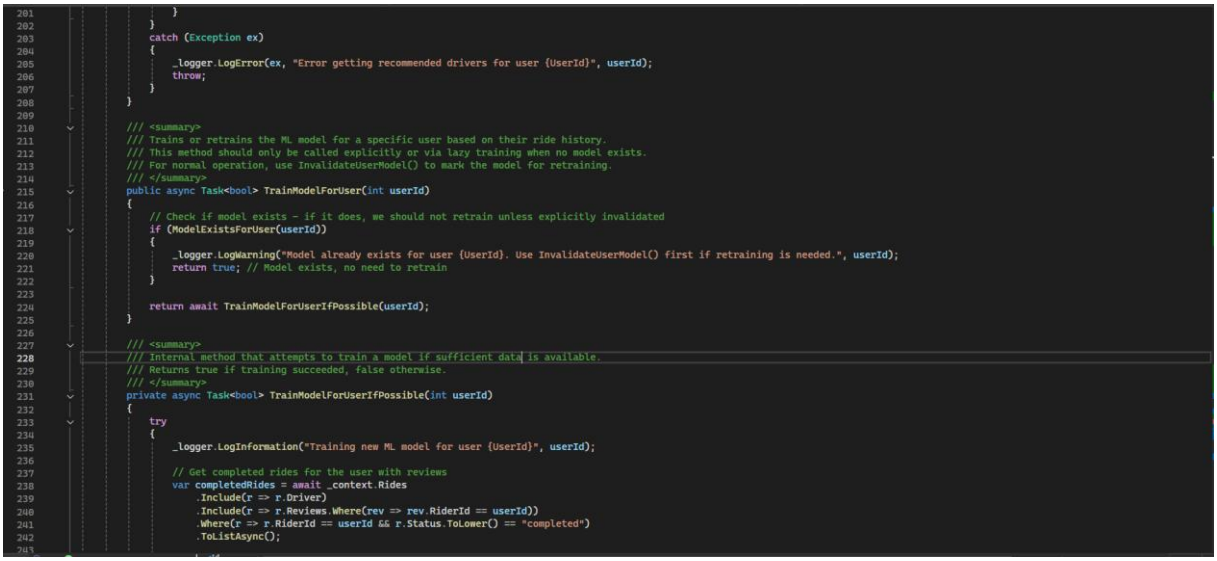
Printscreenovi souce code-a glavne logike recommender sistema



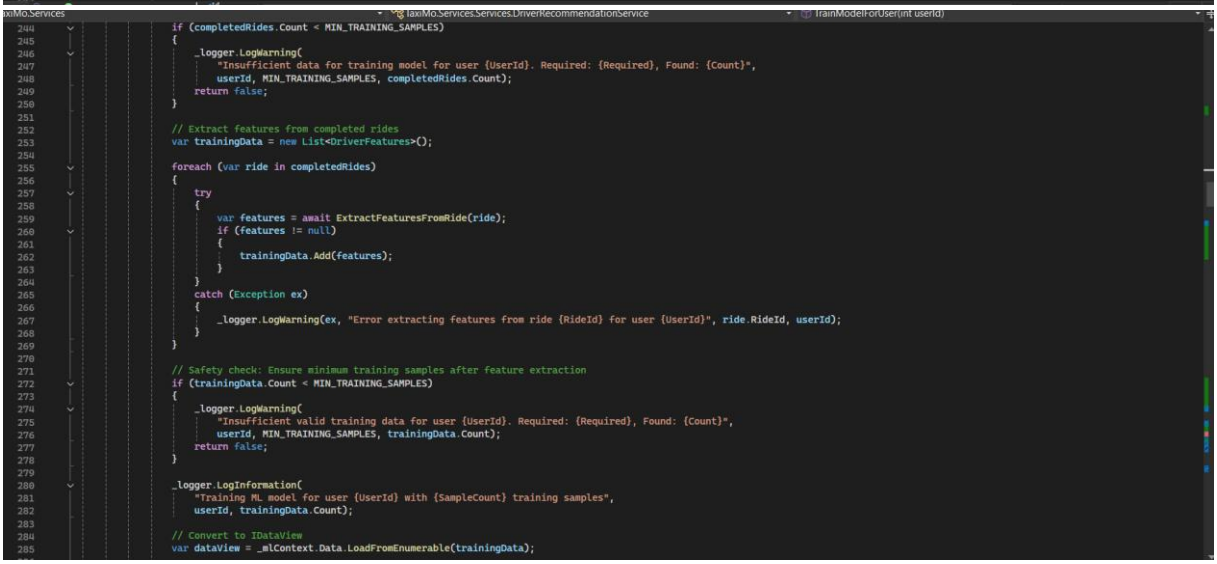
```
/// If a model exists, it loads it from cache and uses it for predictions.
/// </summary>
public async Task<List<DriverDto>> GetRecommendedDriversForUser(int userId, int topN = 5)
{
    try
    {
        _logger.LogInformation("Getting recommended drivers for user {UserId}, topN: {TopN}", userId, topN);

        // Clamp topN to reasonable bounds (min 1, max 20)
        topN = Math.Clamp(topN, 1, 20);

        // Get all available drivers (using the same logic as DriverService.GetFreeDriversAsync)
        var availableDrivers = await GetAvailableDriversAsync();

        if (!availableDrivers.Any())
        {
            _logger.LogWarning("No available drivers found for user {UserId}", userId);
            return new List<DriverDto>();
        }

        // Load user's ride history with drivers for history bonus calculation
        // This includes completed rides with reviews to determine user's experience with each driver
        var userRideHistory = await _context.Rides
            .Include(r => r.Reviews.Where(rev => rev.RiderId == userId))
            .Where(r => r.RiderId == userId)
            .ToListAsync();

        _logger.LogInformation("Found {AvailableCount} available drivers for user {UserId}. User has {HistoryCount} total rides in history.",
            availableDrivers.Count, userId, userRideHistory.Count);

        // LAZY TRAINING: Check if model exists, if not try to train one
        if (!ModelExistsForUser(userId))
        {
            _logger.LogInformation("Model does not exist for user {UserId}, attempting lazy training", userId);

            // Try to train a model if sufficient data exists
            var trainingSuccess = await TrainModelForUserIfPossible(userId);

            if (!trainingSuccess)
            {
                // Not enough data for training - use cold start
                _logger.LogInformation("Insufficient data for training model for user {UserId}, using cold start strategy", userId);
                return await ColdStartRecommendationWithHistory(availableDrivers, userRideHistory, userId, topN);
            }

            // Training succeeded, continue to use the model
            _logger.LogInformation("Successfully trained new ML model for user {UserId}", userId);
        }
        else
        {
            _logger.LogInformation("Loading cached ML model for user {UserId}", userId);
        }

        // Load the model and make predictions
        try
        {
            var modelPath = GetModelPath(userId);
            var mlModel = _mlContext.Model.Load(modelPath, out var modelInputSchema);
            var predictionEngine = _mlContext.Model.CreatePredictionEngine<DriverFeatures, DriverPrediction>(mlModel);

            // Get user's average ride characteristics for prediction
            var userAvgRideStats = await GetUserAverageRideStatsAsync(userId);

            // Predict scores for each available driver and combine with history bonus
            var driverScores = new List<(Driver Driver, float FinalScore, float MlScore, float HistoryBonus)>();

            foreach (var driver in availableDrivers)
            {
                try
                {
                    // Filter out drivers with whom user already had completed rides (unless rating was excellent 4.5+)
                    if (ShouldExcludeDriver(userId, driver.DriverId, userRideHistory))
                    {
                        _logger.LogDebug("Excluding driver {DriverId} for user {UserId} - already had completed ride with lower rating",
                            driver.DriverId, userId);
                        continue;
                    }

                    // Get ML prediction score
                    var features = ExtractFeaturesForDriver(driver, userAvgRideStats);
                    var prediction = predictionEngine.Predict(features);
                    var mlScore = prediction.PredictedScore;

                    // Safety check: Ignore NaN or Infinity predictions
                    if (float.IsNaN(mlScore) || float.IsInfinity(mlScore))
                    {
                        _logger.LogWarning("Invalid ML prediction score (NaN/Infinity) for driver {DriverId}, user {UserId}. Skipping driver.",
                            driver.DriverId, userId);
                        continue;
                    }

                    // Calculate history bonus based on user's previous experience with this driver
                    var historyBonus = CalculateHistoryBonus(userId, driver.DriverId, userRideHistory);

                    // Combine ML score with history bonus: finalScore = mlScore + historyBonus
                    var finalScore = mlScore + historyBonus;

                    driverScores.Add((driver, finalScore, mlScore, historyBonus));

                    // Log when history bonus is applied
                    if (Math.Abs(historyBonus) > 0.01f) // Only log if bonus is significant
                    {
                        _logger.LogInformation(
                            "History bonus applied for driver {DriverId}, user {UserId}: ML Score={MlScore:F3}, History Bonus={HistoryBonus:F3}, Final Score={FinalScore:F3}",
                            driver.DriverId, userId, mlScore, historyBonus, finalScore);
                    }
                }
                catch (Exception ex)
                {
                    _logger.LogWarning(ex, "Error calculating score for driver {DriverId}, user {UserId}. Skipping driver.",
                        driver.DriverId, userId);
                    continue;
                }
            }

            // If no valid predictions remain, fallback to cold start
            if (!driverScores.Any())
            {
                _logger.LogWarning("No valid predictions generated for user {UserId}, falling back to cold start", userId);
                return await ColdStartRecommendationWithHistory(availableDrivers, userRideHistory, userId, topN);
            }
```



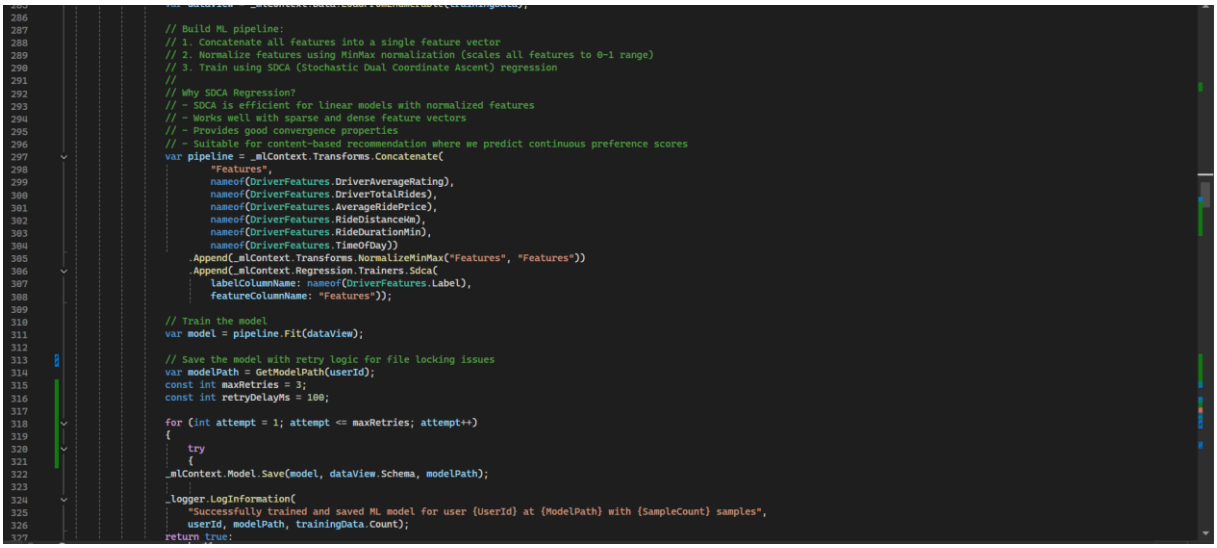
```
                        driver.DriverId, userId);
                    continue;
                }
            }

            // If no valid predictions remain, fallback to cold start
            if (!driverScores.Any())
            {
                _logger.LogWarning("No valid predictions generated for user {UserId}, falling back to cold start", userId);
                return await ColdStartRecommendationWithHistory(availableDrivers, userRideHistory, userId, topN);
            }

            // Sort by final score (ML score + history bonus) descending and take top N
            var topDrivers = driverScores
                .OrderByDescending(x => x.FinalScore)
                .Take(topN)
                .ToList();

            // Map to DTOs with actual rating and rides count
            var recommendedDrivers = new List<DriverDto>();
            foreach (var driverScore in topDrivers)
            {
                var dto = await MapToDriverDtoAsync(driverScore.Driver);
                recommendedDrivers.Add(dto);
            }

            // Log metrics including history bonus information
            var topScores = driverScores.OrderByDescending(x => x.FinalScore).Take(topN).ToList();
            var scoreDetails = topScores.Select(x =>
                $"ML:{x.MlScore:F3}+History:{x.HistoryBonus:F3}={x.FinalScore:F3}").ToList();
            _logger.LogInformation(
                "Returned {Count} recommended drivers for user {UserId}. Top scores: [{Scores}]",
                recommendedDrivers.Count, userId, string.Join(", ", scoreDetails));

            return recommendedDrivers;
        }
        catch (Exception ex)
        {
            _logger.LogWarning(ex, "Error loading or using model for user {UserId}, falling back to cold start", userId);
            return await ColdStartRecommendationWithHistory(availableDrivers, userRideHistory, userId, topN);
        }
    }
    catch (Exception ex)
```



```
            }
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error getting recommended drivers for user {UserId}", userId);
            throw;
        }
    }

    /// <summary>
    /// Trains or retrains the ML model for a specific user based on their ride history.
    /// This method should only be called explicitly or via lazy training when no model exists.
    /// For normal operation, use InvalidateUserModel() to mark the model for retraining.
    /// </summary>
    public async Task<bool> TrainModelForUser(int userId)
    {
        // Check if model exists - if it does, we should not retrain unless explicitly invalidated
        if (ModelExistsForUser(userId))
        {
            _logger.LogWarning("Model already exists for user {UserId}. Use InvalidateUserModel() first if retraining is needed.", userId);
            return true; // Model exists, no need to retrain
        }

        return await TrainModelForUserIfPossible(userId);
    }

    /// <summary>
    /// Internal method that attempts to train a model if sufficient data is available.
    /// Returns true if training succeeded, false otherwise.
    /// </summary>
    private async Task<bool> TrainModelForUserIfPossible(int userId)
    {
        try
        {
            _logger.LogInformation("Training new ML model for user {UserId}", userId);

            // Get completed rides for the user with reviews
            var completedRides = await _context.Rides
                .Include(r => r.Driver)
                .Include(r => r.Reviews.Where(rev => rev.RiderId == userId))
                .Where(r => r.RiderId == userId && r.Status.ToLower() == "completed")
                .ToListAsync();
```



```
TaxiMO.Services    TaxiMO.Services.Services.DriverRecommendationService    TrainModelForUser(int userId)
            if (completedRides.Count < MIN_TRAINING_SAMPLES)
            {
                _logger.LogWarning(
                    "Insufficient data for training model for user {UserId}. Required: {Required}, Found: {Count}",
                    userId, MIN_TRAINING_SAMPLES, completedRides.Count);
                return false;
            }

            // Extract features from completed rides
            var trainingData = new List<DriverFeatures>();

            foreach (var ride in completedRides)
            {
                try
                {
                    var features = await ExtractFeaturesFromRide(ride);
                    if (features != null)
                    {
                        trainingData.Add(features);
                    }
                }
                catch (Exception ex)
                {
                    _logger.LogWarning(ex, "Error extracting features from ride {RideId} for user {UserId}", ride.RideId, userId);
                }
            }

            // Safety check: Ensure minimum training samples after feature extraction
            if (trainingData.Count < MIN_TRAINING_SAMPLES)
            {
                _logger.LogWarning(
                    "Insufficient valid training data for user {UserId}. Required: {Required}, Found: {Count}",
                    userId, MIN_TRAINING_SAMPLES, trainingData.Count);
                return false;
            }

            _logger.LogInformation(
                "Training ML model for user {UserId} with {SampleCount} training samples",
                userId, trainingData.Count);

            // Convert to IDataView
            var dataView = _mlContext.Data.LoadFromEnumerable(trainingData);
```



```
286
287             // Build ML pipeline:
288             // 1. Concatenate all features into a single feature vector
289             // 2. Normalize features using MinMax normalization (scales all features to 0-1 range)
290             // 3. Train using SDCA (Stochastic Dual Coordinate Ascent) regression
291             //
292             // Why SDCA Regression?
293             // - SDCA is efficient for linear models with normalized features
294             // - Works well with sparse and dense feature vectors
295             // - Provides good convergence properties
296             // - Suitable for content-based recommendation where we predict continuous preference scores
297             var pipeline = _mlContext.Transforms.Concatenate(
298                     "Features",
299                     nameof(DriverFeatures.DriverAverageRating),
300                     nameof(DriverFeatures.DriverTotalRides),
301                     nameof(DriverFeatures.AverageRidePrice),
302                     nameof(DriverFeatures.RideDistanceKm),
303                     nameof(DriverFeatures.RideDurationMin),
304                     nameof(DriverFeatures.TimeOfDay))
305                 .Append(_mlContext.Transforms.NormalizeMinMax("Features", "Features"))
306                 .Append(_mlContext.Regression.Trainers.Sdca(
307                     labelColumnName: nameof(DriverFeatures.Label),
308                     featureColumnName: "Features"));
309
310             // Train the model
311             var model = pipeline.Fit(dataView);
312
313             // Save the model with retry logic for file locking issues
314             var modelPath = GetModelPath(userId);
315             const int maxRetries = 3;
316             const int retryDelayMs = 100;
317
318             for (int attempt = 1; attempt <= maxRetries; attempt++)
319             {
320                 try
321                 {
322             _mlContext.Model.Save(model, dataView.Schema, modelPath);
323
324             _logger.LogInformation(
325                 "Successfully trained and saved ML model for user {UserId} at {ModelPath} with {SampleCount} samples",
326                 userId, modelPath, trainingData.Count);
327             return true;
```

Putanja do code-a u aplikaciji gdje se poziva recommender sistem:

TaxiMo/TaxiMo/TaxiMoUI/taximo_mobile/lib/user/screens/user_home_screen.dart

Printscreen iz pokrenute aplikacije:

Welcome back 👋

Ready for your next ride?

Book a Ride

Choose pickup & destination in seconds

Recommended Drivers

★ 4.7

Driver Driver

14 rides

Amina Kovacevic

8 rides

Quick Actions

Trip History

Payments