Amina Gutošić

Razvoj softvera 2

01.11.2025.

Implementacija recommender sistema

ScoutTrack je društvena aplikacija za izviđače koja im omogućava da se prijavljuju na aktivnosti, povezuju sa drugim korisnicima i komuniciraju putem objava i komentara. Kako aplikacija raste, a broj aktivnosti i korisnika postaje sve veći, korisnicima postaje teže da pronađu nove i zanimljive aktivnosti koje odgovaraju njihovim interesima. Zato recommender sistem ima važnu ulogu – on personalizuje iskustvo svakog korisnika tako što predlaže aktivnosti koje bi mu se najvjerovatnije svidjele i potencijalne prijatelje sa kojima dijeli slične izviđačke interese.

Unutar ove aplikacije implementirana su dva sistema preporuke: preporuka aktivnosti i preporuka prijatelja.

<u>Preporuka aktivnosti</u>

Da bi korisnik dobio personalizovane preporuke, sistem koristi kombinaciju globalnog modela mašinskog učenja i ličnih preferencija korisnika.

1. Učitavanje modela

Prilikom prvog pokretanja, sistem provjerava da li postoji već istrenirani model (Models/GlobalActivityModel.zip):

- Ako postoji – model se učitava i koristi odmah.
- Ako ne postoji – sistem automatski trenira novi model na osnovu historijskih podataka.

2. Priprema podataka

Model uči na osnovu stvarnih podataka o aktivnostima koje su korisnici završili ili ocijenili visokom ocjenom (recenzije).

Za svaku aktivnost uzimaju se sljedeće karakteristike:

- Geografska lokacija (latitude i longitude)
- Tip aktivnosti (npr. kampovanje, sport, edukacija…)
- Odred koji je organizovao aktivnost
- Kotizacija (fee)

- Trajanje aktivnosti u satima
- Mjesec u kojem se aktivnost održava

Ovi podaci se koriste za treniranje regresijskog modela pomoću SDCA algoritma (Stochastic Dual Coordinate Ascent), koji uči obrasce između sadržaja aktivnosti i pozitivnih interakcija korisnika.

3. Treniranje i čuvanje modela

Model se trenira jednom i zatim se sprema u datoteku GlobalActivityModel.zip. Pri svakom narednom pozivu koristi se taj spremljeni model dok se ne regeneriše.

4. Predikcija i izračunavanje score-a

Za svakog korisnika, sistem:

1. Pronalazi aktivnosti koje su otvorene za registraciju/prijavu, a na koje se korisnik još nije prijavio.
2. Filtrira privatne aktivnosti – prikazuju se samo one koje organizuje odred kojem korisnik pripada (javne aktivnosti su svakako dostupne svima).
3. Za svaku od preostalih aktivnosti izračunava ukupni **score** kombinovanjem:
    - Globalnog modela (60%) – na osnovu karakteristika same aktivnosti.
    - Ličnih preferencija korisnika (40%) – na osnovu njegovih prethodnih aktivnosti i recenzija.

5. Lične preferencije

Sistem analizira:

- Do tri najčešće vrste aktivnosti kojima korisnik prisustvuje i/ili ocjenjuje visokom ocjenom.
- Prosječnu cijenu i trajanje aktivnosti na kojima učestvuje.
- Lokacije aktivnosti koje najčešće posjećuje.

Na osnovu toga se računa **lični score** koji odražava koliko se nova aktivnost uklapa u obrasce ponašanja korisnika.

6. Filtriranje i prikaz

Samo aktivnosti koje:

- imaju status *"RegistrationsOpenActivityState"*,
- na koje se korisnik već nije prijavio,
- i (ako su privatne) pripadaju njegovom odredu (troop-u),

ulaze u konačni izbor.

Rezultat je lista Top-N preporuka koje se prikazuju korisniku.

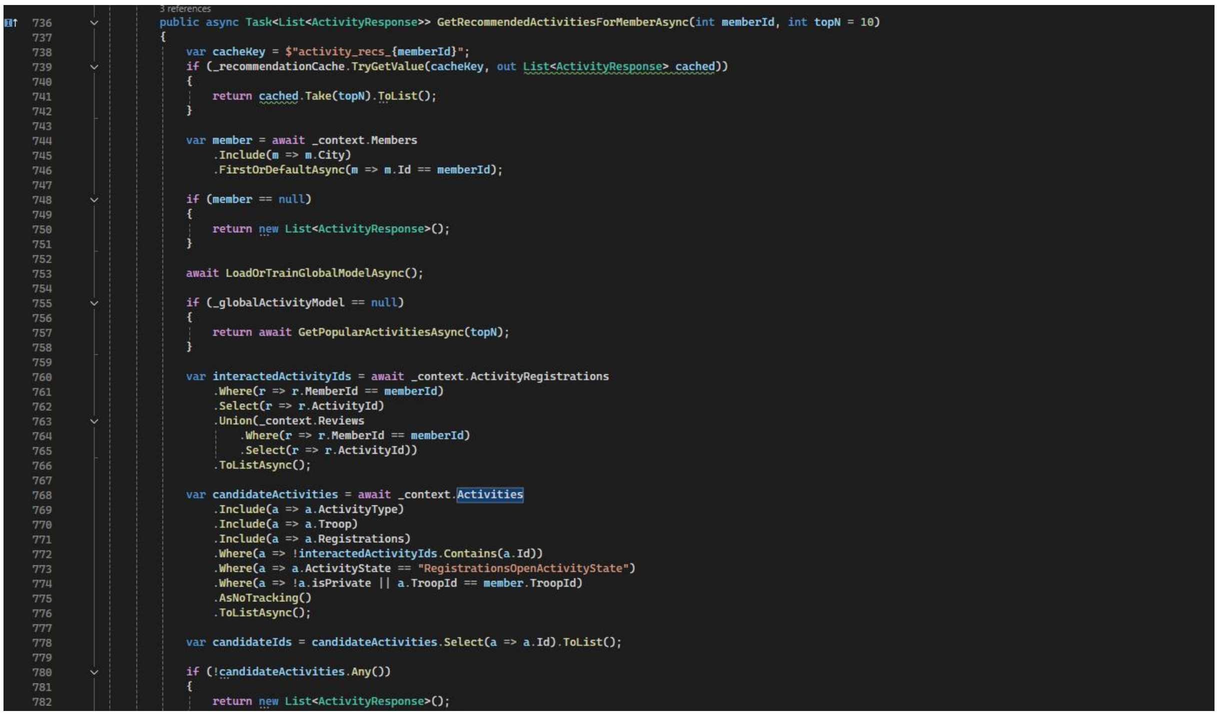
7. Cold Start – novi korisnici

Ako korisnik nema dovoljno podataka, sistem mu prikazuje najpopularnije aktivnosti (one sa najviše prijava).

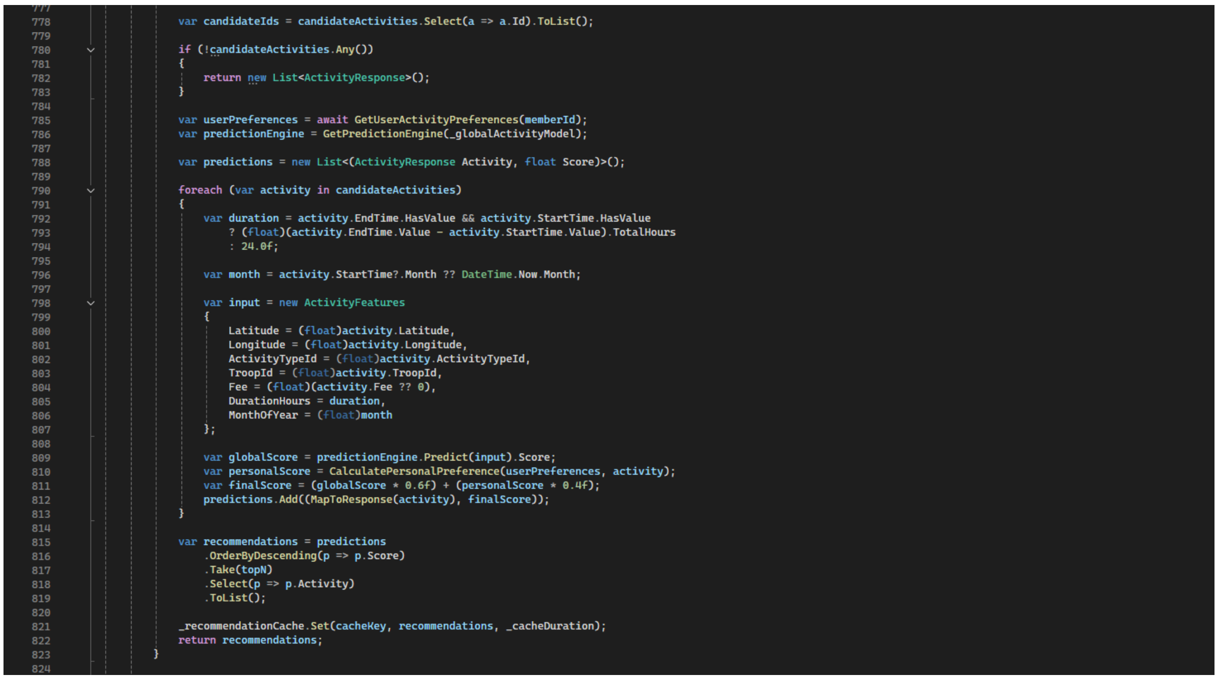
Putanja do source code-a:

ScoutTrack\ScoutTrack\ScoutTrack.Services\Services\ActivityService.cs

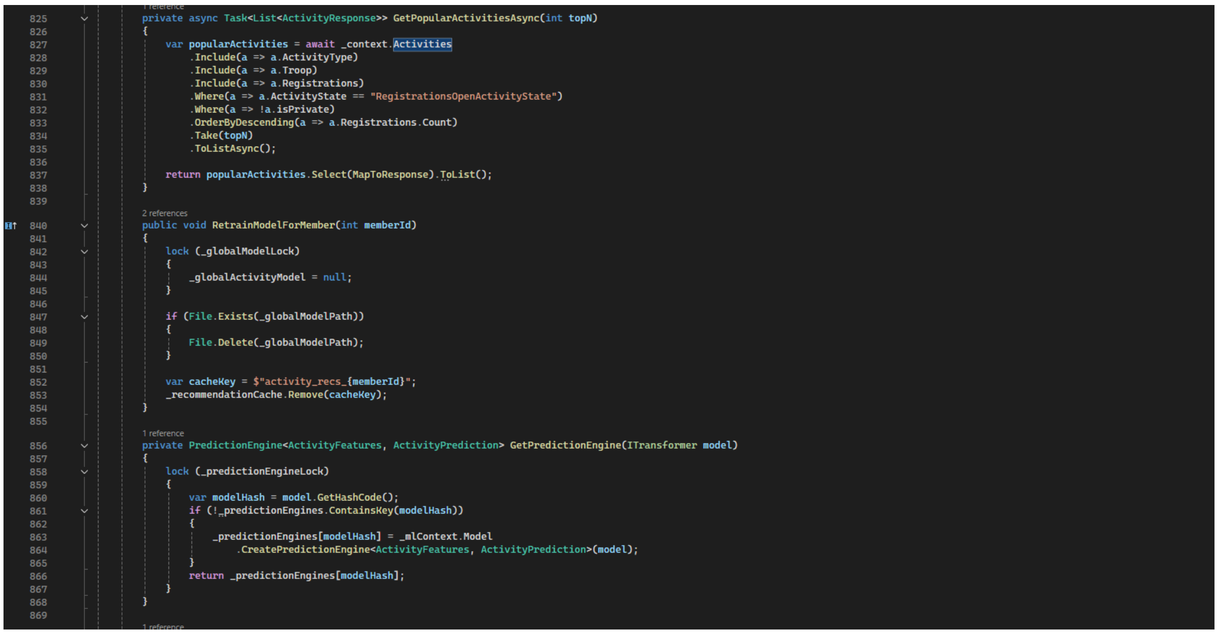
Printscreenovi source code-a glavne logike activity recommender sistema:



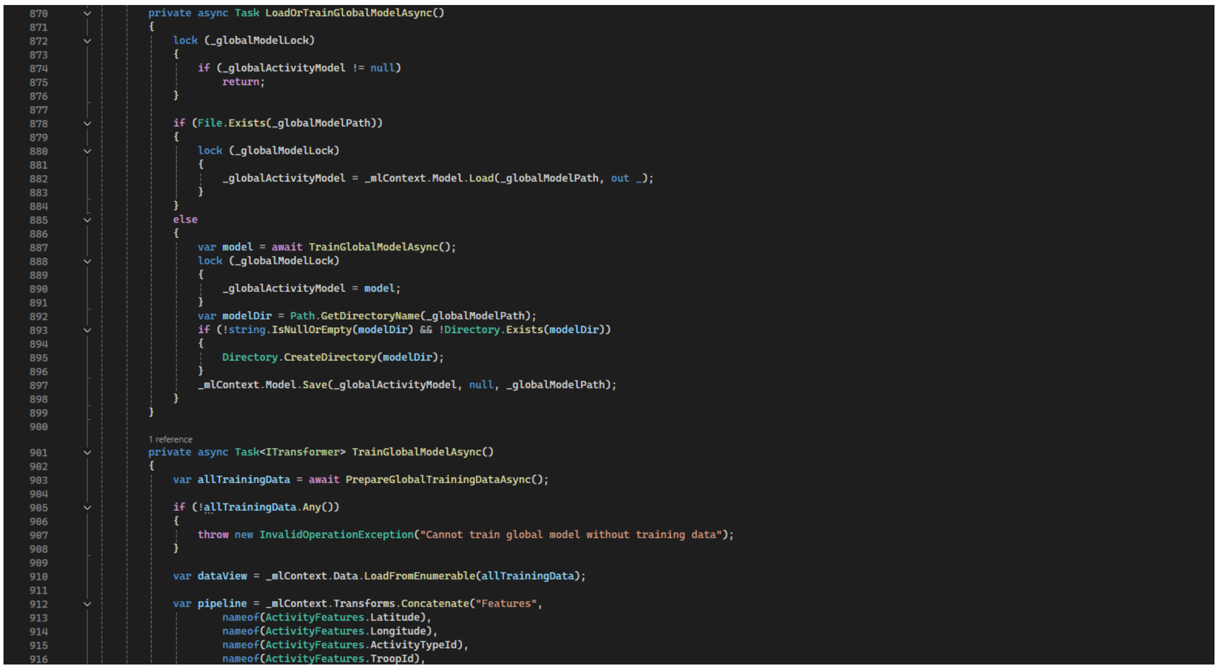
```
3 references
public async Task<List<ActivityResponse>> GetRecommendedActivitiesForMemberAsync(int memberId, int topN = 10)
{
    var cacheKey = $"activity_recs_{memberId}";
    if (_recommendationCache.TryGetValue(cacheKey, out List<ActivityResponse> cached))
    {
        return cached.Take(topN).ToList();
    }

    var member = await _context.Members
        .Include(m => m.City)
        .FirstOrDefaultAsync(m => m.Id == memberId);

    if (member == null)
    {
        return new List<ActivityResponse>();
    }

    await LoadOrTrainGlobalModelAsync();

    if (_globalActivityModel == null)
    {
        return await GetPopularActivitiesAsync(topN);
    }

    var interactedActivityIds = await _context.ActivityRegistrations
        .Where(r => r.MemberId == memberId)
        .Select(r => r.ActivityId)
        .Union(_context.Reviews
            .Where(r => r.MemberId == memberId)
            .Select(r => r.ActivityId))
        .ToListAsync();

    var candidateActivities = await _context.Activities
        .Include(a => a.ActivityType)
        .Include(a => a.Troop)
        .Include(a => a.Registrations)
        .Where(a => !interactedActivityIds.Contains(a.Id))
        .Where(a => a.ActivityState == "RegistrationsOpenActivityState")
        .Where(a => !a.isPrivate || a.TroopId == member.TroopId)
        .AsNoTracking()
        .ToListAsync();

    var candidateIds = candidateActivities.Select(a => a.Id).ToList();

    if (!candidateActivities.Any())
    {
        return new List<ActivityResponse>();
```



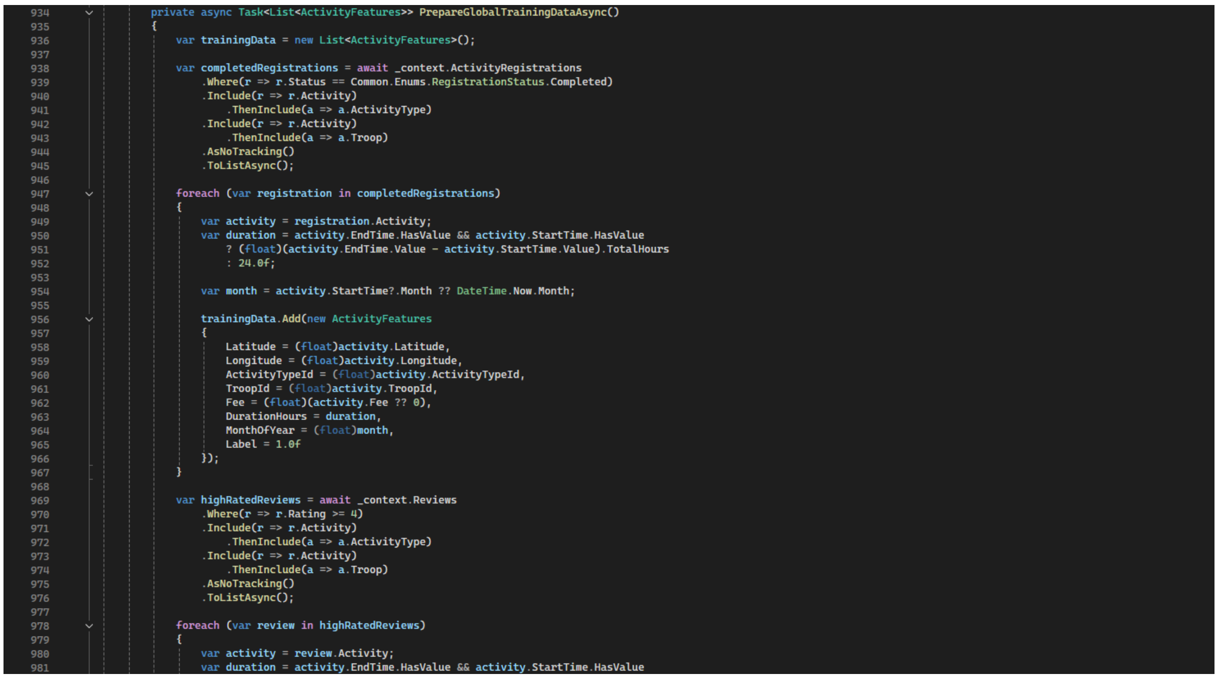
```csharp
            var candidateIds = candidateActivities.Select(a => a.Id).ToList();

            if (!candidateActivities.Any())
            {
                return new List<ActivityResponse>();
            }

            var userPreferences = await GetUserActivityPreferences(memberId);
            var predictionEngine = GetPredictionEngine(_globalActivityModel);

            var predictions = new List<(ActivityResponse Activity, float Score)>();

            foreach (var activity in candidateActivities)
            {
                var duration = activity.EndTime.HasValue && activity.StartTime.HasValue
                    ? (float)(activity.EndTime.Value - activity.StartTime.Value).TotalHours
                    : 24.0f;

                var month = activity.StartTime?.Month ?? DateTime.Now.Month;

                var input = new ActivityFeatures
                {
                    Latitude = (float)activity.Latitude,
                    Longitude = (float)activity.Longitude,
                    ActivityTypeId = (float)activity.ActivityTypeId,
                    TroopId = (float)activity.TroopId,
                    Fee = (float)(activity.Fee ?? 0),
                    DurationHours = duration,
                    MonthOfYear = (float)month
                };

                var globalScore = predictionEngine.Predict(input).Score;
                var personalScore = CalculatePersonalPreference(userPreferences, activity);
                var finalScore = (globalScore * 0.6f) + (personalScore * 0.4f);
                predictions.Add((MapToResponse(activity), finalScore));
            }

            var recommendations = predictions
                .OrderByDescending(p => p.Score)
                .Take(topN)
                .Select(p => p.Activity)
                .ToList();

            _recommendationCache.Set(cacheKey, recommendations, _cacheDuration);
            return recommendations;
        }

```



```csharp
        1 reference
        private async Task<List<ActivityResponse>> GetPopularActivitiesAsync(int topN)
        {
            var popularActivities = await _context.Activities
                .Include(a => a.ActivityType)
                .Include(a => a.Troop)
                .Include(a => a.Registrations)
                .Where(a => a.ActivityState == "RegistrationsOpenActivityState")
                .Where(a => !a.isPrivate)
                .OrderByDescending(a => a.Registrations.Count)
                .Take(topN)
                .ToListAsync();

            return popularActivities.Select(MapToResponse).ToList();
        }

        2 references
        public void RetrainModelForMember(int memberId)
        {
            lock (_globalModelLock)
            {
                _globalActivityModel = null;
            }

            if (File.Exists(_globalModelPath))
            {
                File.Delete(_globalModelPath);
            }

            var cacheKey = $"activity_recs_{memberId}";
            _recommendationCache.Remove(cacheKey);
        }

        1 reference
        private PredictionEngine<ActivityFeatures, ActivityPrediction> GetPredictionEngine(ITransformer model)
        {
            lock (_predictionEngineLock)
            {
                var modelHash = model.GetHashCode();
                if (!_predictionEngines.ContainsKey(modelHash))
                {
                    _predictionEngines[modelHash] = _mlContext.Model
                        .CreatePredictionEngine<ActivityFeatures, ActivityPrediction>(model);
                }
                return _predictionEngines[modelHash];
            }
        }

        1 reference
```



```csharp
private async Task LoadOrTrainGlobalModelAsync()
{
    lock (_globalModelLock)
    {
        if (_globalActivityModel != null)
            return;
    }

    if (File.Exists(_globalModelPath))
    {
        lock (_globalModelLock)
        {
            _globalActivityModel = _mlContext.Model.Load(_globalModelPath, out _);
        }
    }
    else
    {
        var model = await TrainGlobalModelAsync();
        lock (_globalModelLock)
        {
            _globalActivityModel = model;
        }
        var modelDir = Path.GetDirectoryName(_globalModelPath);
        if (!string.IsNullOrEmpty(modelDir) && !Directory.Exists(modelDir))
        {
            Directory.CreateDirectory(modelDir);
        }
        _mlContext.Model.Save(_globalActivityModel, null, _globalModelPath);
    }
}

1 reference
private async Task<ITransformer> TrainGlobalModelAsync()
{
    var allTrainingData = await PrepareGlobalTrainingDataAsync();

    if (!allTrainingData.Any())
    {
        throw new InvalidOperationException("Cannot train global model without training data");
    }

    var dataView = _mlContext.Data.LoadFromEnumerable(allTrainingData);

    var pipeline = _mlContext.Transforms.Concatenate("Features",
            nameof(ActivityFeatures.Latitude),
            nameof(ActivityFeatures.Longitude),
            nameof(ActivityFeatures.ActivityTypeId),
            nameof(ActivityFeatures.TroopId),
```

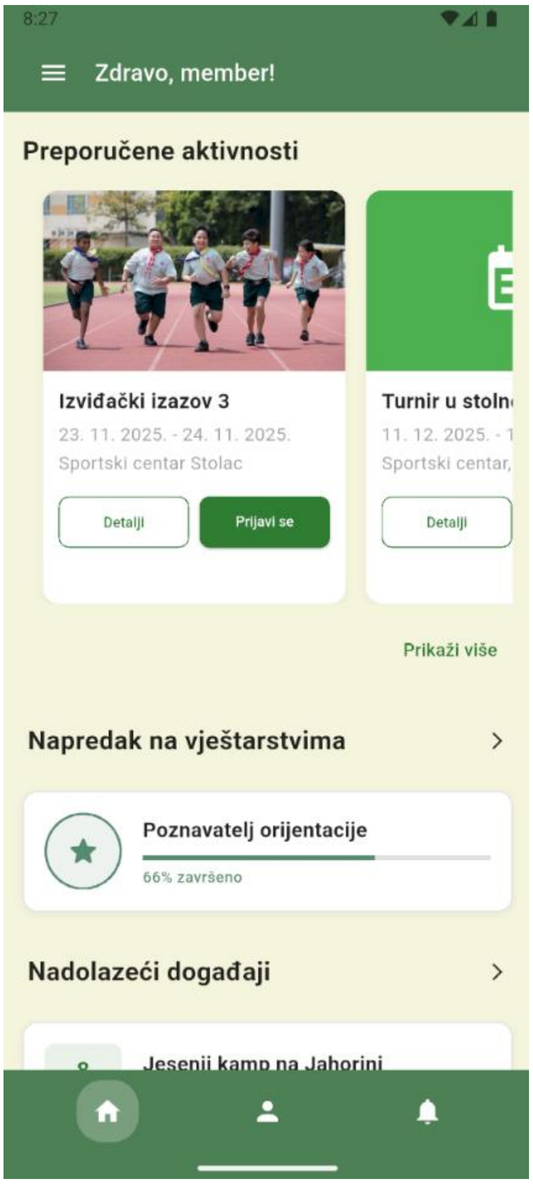


```csharp
private async Task<List<ActivityFeatures>> PrepareGlobalTrainingDataAsync()
{
    var trainingData = new List<ActivityFeatures>();

    var completedRegistrations = await _context.ActivityRegistrations
        .Where(r => r.Status == Common.Enums.RegistrationStatus.Completed)
        .Include(r => r.Activity)
            .ThenInclude(a => a.ActivityType)
        .Include(r => r.Activity)
            .ThenInclude(a => a.Troop)
        .AsNoTracking()
        .ToListAsync();

    foreach (var registration in completedRegistrations)
    {
        var activity = registration.Activity;
        var duration = activity.EndTime.HasValue && activity.StartTime.HasValue
            ? (float)(activity.EndTime.Value - activity.StartTime.Value).TotalHours
            : 24.0f;

        var month = activity.StartTime?.Month ?? DateTime.Now.Month;

        trainingData.Add(new ActivityFeatures
        {
            Latitude = (float)activity.Latitude,
            Longitude = (float)activity.Longitude,
            ActivityTypeId = (float)activity.ActivityTypeId,
            TroopId = (float)activity.TroopId,
            Fee = (float)(activity.Fee ?? 0),
            DurationHours = duration,
            MonthOfYear = (float)month,
            Label = 1.0f
        });
    }

    var highRatedReviews = await _context.Reviews
        .Where(r => r.Rating >= 4)
        .Include(r => r.Activity)
            .ThenInclude(a => a.ActivityType)
        .Include(r => r.Activity)
            .ThenInclude(a => a.Troop)
        .AsNoTracking()
        .ToListAsync();

    foreach (var review in highRatedReviews)
    {
        var activity = review.Activity;
        var duration = activity.EndTime.HasValue && activity.StartTime.HasValue
```

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

ScoutTrack\ScoutTrack\ScoutTrack.UI\scouttrack_mobile\lib\screens\home_screen.dart

Printscreen iz pokrenute aplikacije:



Preporuka prijatelja

Sistem koristi **kolaborativno filtriranje** zasnovano na **matričnoj faktorizaciji (Matrix Factorization)**. Ideja je da se korisnici koji imaju slične obrasce ponašanja (npr. učestvuju u istim aktivnostima, lajkuju iste objave ili komentarišu sličan sadržaj) smatraju **sličnima** i da se međusobno preporučuju.

Model se trenira koristeći **ML.NET MatrixFactorizationTrainer**, koji predviđa *score* sličnosti između korisnika.

1. Priprema podataka za treniranje

Sistem automatski generiše podatke za treniranje na osnovu stvarnih korisničkih interakcija.

A) Zajednički interesi (indirektna sličnost)

- Zajedničke aktivnosti (Common Activities)
- Zajednički lajkovi (Shared Likes)
- Zajednički komentari (Shared Comments)
- Zajedničke recenzije (Common Reviews)

B) Direktne interakcije (međusobna angažovanost)

- Korisnik A lajkuje objave korisnika B
- Korisnik B lajkuje objave korisnika A
- Korisnik A komentariše objave korisnika B
- Korisnik B komentariše objave korisnika A

Svaka od ovih interakcija ima svoju težinu (ponder), gdje direktne interakcije nose veću važnost jer ukazuju na jaču društvenu povezanost. Na osnovu ukupnih interakcija računa se **normalizovani score sličnosti (0–1)**.

2. Treniranje modela

Ako model FriendRecommendationModel.zip već postoji, sistem ga učitava i koristi odmah. Ako ne postoji, model se automatski trenira pomoću stvarnih korisničkih podataka. Model se sprema nakon treniranja i koristi za buduće preporuke bez ponovnog treniranja.

Sistem koristi adaptivne parametre:

- Broj iteracija i rank modela automatski se podešavaju prema broju korisnika u bazi.
- Model se trenira asinhrono i zapisuje u log datoteku.

3. Proces generisanja preporuka

Kada korisnik zatraži preporuke prijatelja:

1. Sistem provjerava **cache** da li već postoji generisana lista preporuka za tog korisnika (u zadnjih 5 minuta).
2. Ako nema dovoljan broj interakcija (tzv. *cold start*), korisniku se prikazuju **najaktivniji članovi** platforme.

3. Ako korisnik ima dovoljno podataka, sistem koristi trenirani ML model za izračunavanje sličnosti.
4. Rezultati se sortiraju po **SimilarityScore** i vraća se Top-N lista preporučenih članova.
5. Postojeći prijatelji, korisnici koji su poslali zahtjev ili kojima je poslan zahtjev od strane ulogovanog korisnika te sami korisnik se automatski isključuju iz rezultata.

4. Cold Start – novi korisnici

Za korisnike koji nemaju dovoljno aktivnosti ili interakcija, sistem prelazi na tzv. **fallback logiku** i predlaže najaktivnije korisnike prema bodovanju:

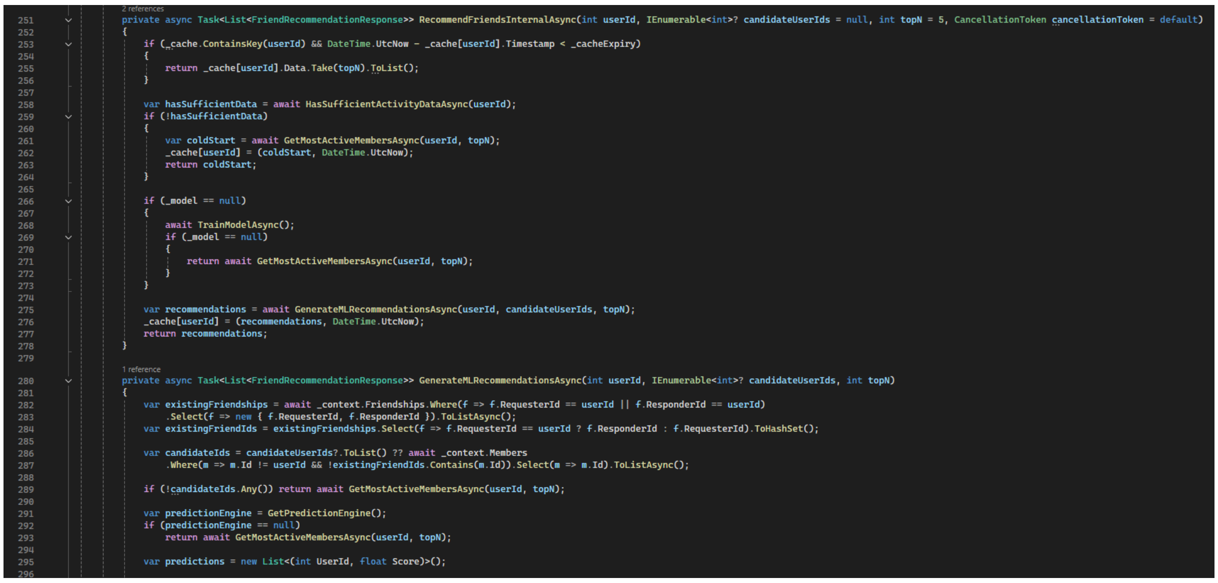
Aktivnost	Bodovi
Odobrene prijave na aktivnosti	2.0
Objave	1.5
Komentari	1.0
Lajkovi	0.5
Recenzije	2.0

Korisnici se zatim sortiraju po ukupnom broju bodova, a najaktivniji ulaze u listu preporučenih prijatelja.

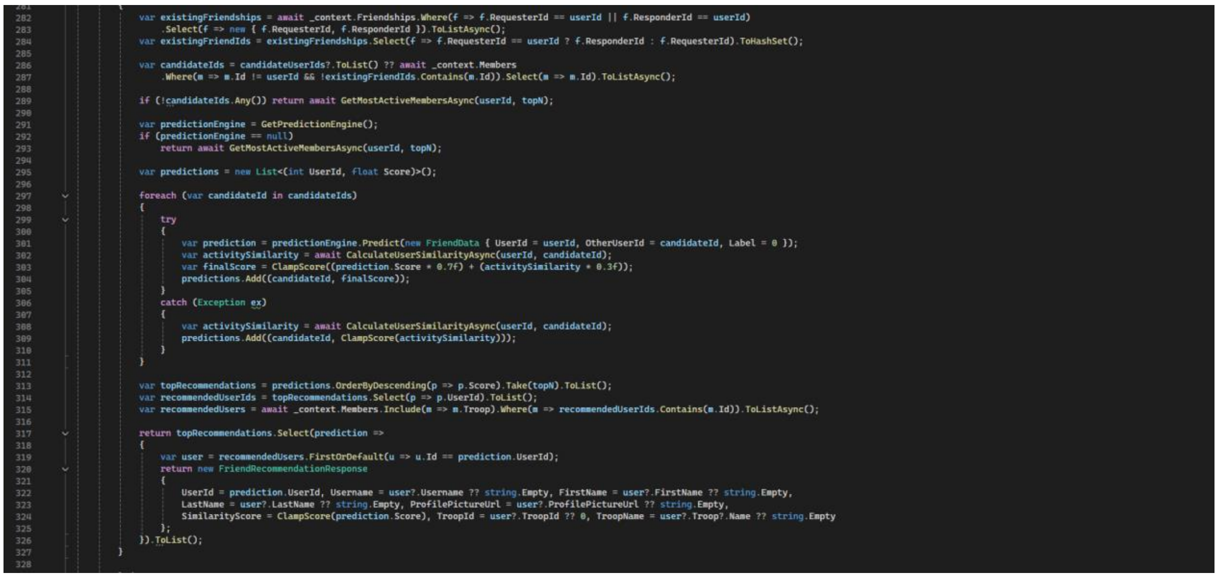
Putanja do source code-a:

ScoutTrack\ScoutTrack\ScoutTrack.Services\Services\FriendshipService.cs

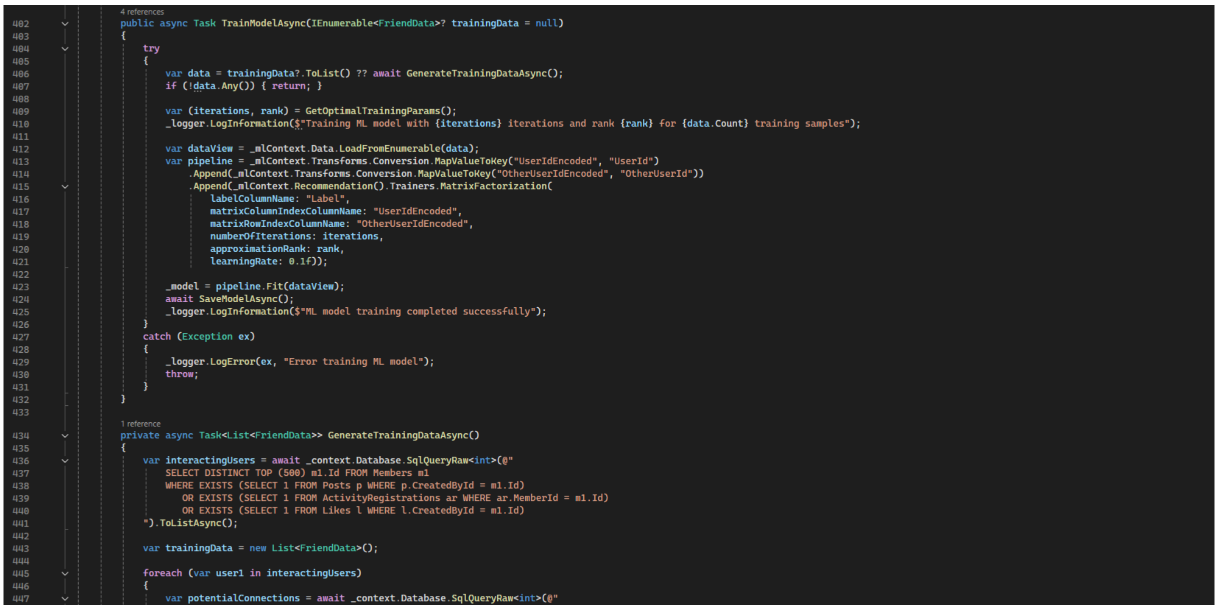
Printscreenovi source code-a glavne logike friend recommender sistema:



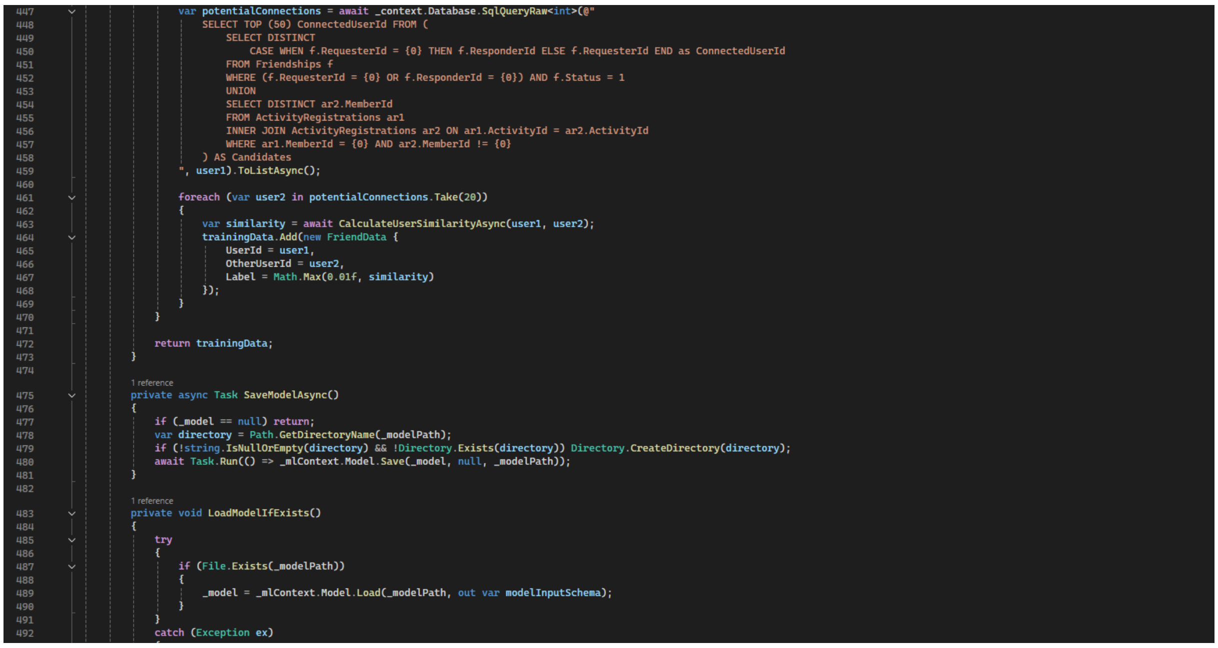
```
2 references
private async Task<List<FriendRecommendationResponse>> RecommendFriendsInternalAsync(int userId, IEnumerable<int>? candidateUserIds = null, int topN = 5, CancellationToken cancellationToken = default)
{
    if (_cache.ContainsKey(userId) && DateTime.UtcNow - _cache[userId].Timestamp < _cacheExpiry)
    {
        return _cache[userId].Data.Take(topN).ToList();
    }

    var hasSufficientData = await HasSufficientActivityDataAsync(userId);
    if (!hasSufficientData)
    {
        var coldStart = await GetMostActiveMembersAsync(userId, topN);
        _cache[userId] = (coldStart, DateTime.UtcNow);
        return coldStart;
    }

    if (_model == null)
    {
        await TrainModelAsync();
        if (_model == null)
        {
            return await GetMostActiveMembersAsync(userId, topN);
        }
    }

    var recommendations = await GenerateMLRecommendationsAsync(userId, candidateUserIds, topN);
    _cache[userId] = (recommendations, DateTime.UtcNow);
    return recommendations;
}

1 reference
private async Task<List<FriendRecommendationResponse>> GenerateMLRecommendationsAsync(int userId, IEnumerable<int>? candidateUserIds, int topN)
{
    var existingFriendships = await _context.Friendships.Where(f => f.RequesterId == userId || f.ResponderId == userId)
        .Select(f => new { f.RequesterId, f.ResponderId }).ToListAsync();
    var existingFriendIds = existingFriendships.Select(f => f.RequesterId == userId ? f.ResponderId : f.RequesterId).ToHashSet();

    var candidateIds = candidateUserIds?.ToList() ?? await _context.Members
        .Where(m => m.Id != userId && !existingFriendIds.Contains(m.Id)).Select(m => m.Id).ToListAsync();

    if (!candidateIds.Any()) return await GetMostActiveMembersAsync(userId, topN);

    var predictionEngine = GetPredictionEngine();
    if (predictionEngine == null)
        return await GetMostActiveMembersAsync(userId, topN);

    var predictions = new List<(int UserId, float Score)>();

```



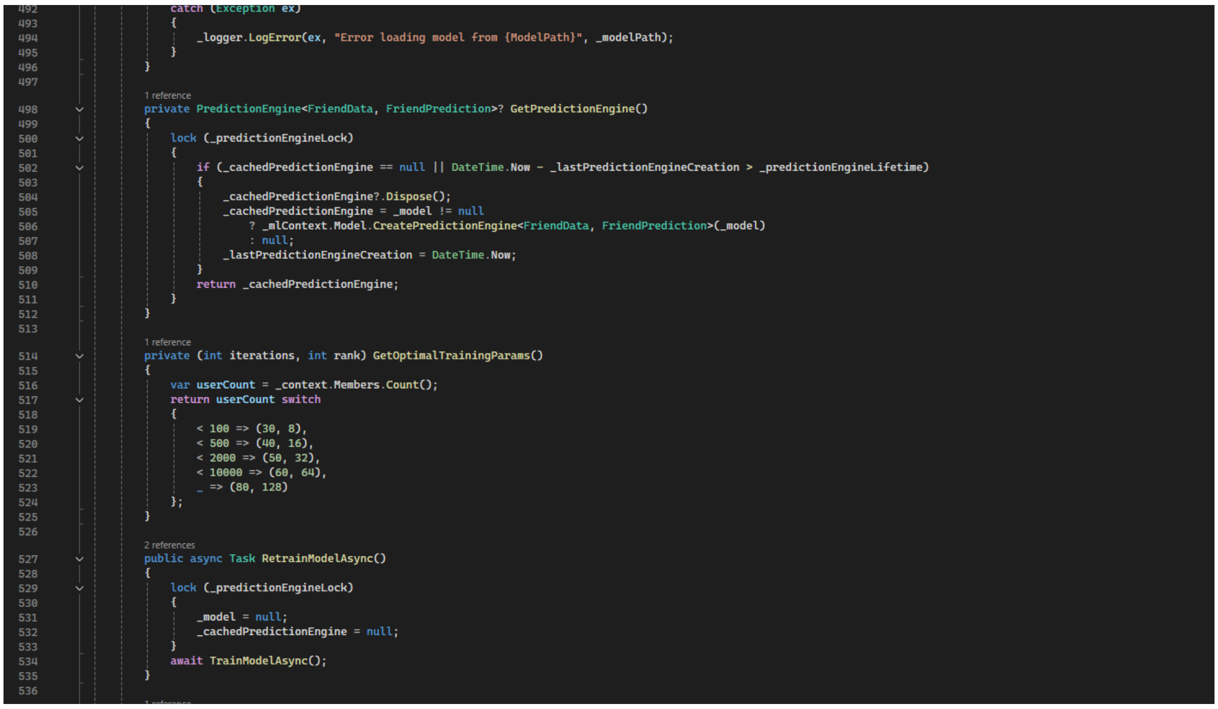
```csharp
{
    var existingFriendships = await _context.Friendships.Where(f => f.RequesterId == userId || f.ResponderId == userId)
        .Select(f => new { f.RequesterId, f.ResponderId }).ToListAsync();
    var existingFriendIds = existingFriendships.Select(f => f.RequesterId == userId ? f.ResponderId : f.RequesterId).ToHashSet();

    var candidateIds = candidateUserIds?.ToList() ?? await _context.Members
        .Where(m => m.Id != userId && !existingFriendIds.Contains(m.Id)).Select(m => m.Id).ToListAsync();

    if (!candidateIds.Any()) return await GetMostActiveMembersAsync(userId, topN);

    var predictionEngine = GetPredictionEngine();
    if (predictionEngine == null)
        return await GetMostActiveMembersAsync(userId, topN);

    var predictions = new List<(int UserId, float Score)>();

    foreach (var candidateId in candidateIds)
    {
        try
        {
            var prediction = predictionEngine.Predict(new FriendData { UserId = userId, OtherUserId = candidateId, Label = 0 });
            var activitySimilarity = await CalculateUserSimilarityAsync(userId, candidateId);
            var finalScore = ClampScore((prediction.Score * 0.7f) + (activitySimilarity * 0.3f));
            predictions.Add((candidateId, finalScore));
        }
        catch (Exception ex)
        {
            var activitySimilarity = await CalculateUserSimilarityAsync(userId, candidateId);
            predictions.Add((candidateId, ClampScore(activitySimilarity)));
        }
    }

    var topRecommendations = predictions.OrderByDescending(p => p.Score).Take(topN).ToList();
    var recommendedUserIds = topRecommendations.Select(p => p.UserId).ToList();
    var recommendedUsers = await _context.Members.Include(m => m.Troop).Where(m => recommendedUserIds.Contains(m.Id)).ToListAsync();

    return topRecommendations.Select(prediction =>
    {
        var user = recommendedUsers.FirstOrDefault(u => u.Id == prediction.UserId);
        return new FriendRecommendationResponse
        {
            UserId = prediction.UserId, Username = user?.Username ?? string.Empty, FirstName = user?.FirstName ?? string.Empty,
            LastName = user?.LastName ?? string.Empty, ProfilePictureUrl = user?.ProfilePictureUrl ?? string.Empty,
            SimilarityScore = ClampScore(prediction.Score), TroopId = user?.TroopId ?? 0, TroopName = user?.Troop?.Name ?? string.Empty
        };
    }).ToList();
}
```



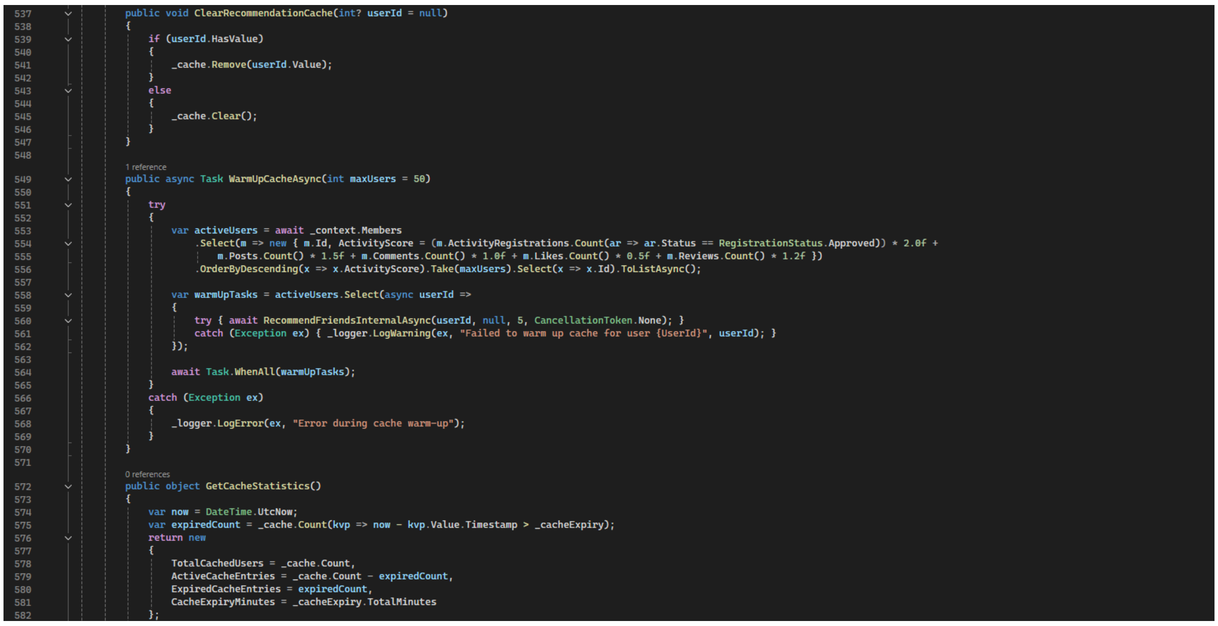
```csharp
4 references
public async Task TrainModelAsync(IEnumerable<FriendData>? trainingData = null)
{
    try
    {
        var data = trainingData?.ToList() ?? await GenerateTrainingDataAsync();
        if (!data.Any()) { return; }

        var (iterations, rank) = GetOptimalTrainingParams();
        _logger.LogInformation($"Training ML model with {iterations} iterations and rank {rank} for {data.Count} training samples");

        var dataView = _mlContext.Data.LoadFromEnumerable(data);
        var pipeline = _mlContext.Transforms.Conversion.MapValueToKey("UserIdEncoded", "UserId")
            .Append(_mlContext.Transforms.Conversion.MapValueToKey("OtherUserIdEncoded", "OtherUserId"))
            .Append(_mlContext.Recommendation().Trainers.MatrixFactorization(
                labelColumnName: "Label",
                matrixColumnIndexColumnName: "UserIdEncoded",
                matrixRowIndexColumnName: "OtherUserIdEncoded",
                numberOfIterations: iterations,
                approximationRank: rank,
                learningRate: 0.1f));

        _model = pipeline.Fit(dataView);
        await SaveModelAsync();
        _logger.LogInformation($"ML model training completed successfully");
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Error training ML model");
        throw;
    }
}

1 reference
private async Task<List<FriendData>> GenerateTrainingDataAsync()
{
    var interactingUsers = await _context.Database.SqlQueryRaw<int>(@"
        SELECT DISTINCT TOP (500) m1.Id FROM Members m1
        WHERE EXISTS (SELECT 1 FROM Posts p WHERE p.CreatedById = m1.Id)
           OR EXISTS (SELECT 1 FROM ActivityRegistrations ar WHERE ar.MemberId = m1.Id)
           OR EXISTS (SELECT 1 FROM Likes l WHERE l.CreatedById = m1.Id)
    ").ToListAsync();

    var trainingData = new List<FriendData>();

    foreach (var user1 in interactingUsers)
    {
        var potentialConnections = await _context.Database.SqlQueryRaw<int>(@"
```



```csharp
        var potentialConnections = await _context.Database.SqlQueryRaw<int>(@"
            SELECT TOP (50) ConnectedUserId FROM (
                SELECT DISTINCT
                    CASE WHEN f.RequesterId = {0} THEN f.ResponderId ELSE f.RequesterId END as ConnectedUserId
                FROM Friendships f
                WHERE (f.RequesterId = {0} OR f.ResponderId = {0}) AND f.Status = 1
                UNION
                SELECT DISTINCT ar2.MemberId
                FROM ActivityRegistrations ar1
                INNER JOIN ActivityRegistrations ar2 ON ar1.ActivityId = ar2.ActivityId
                WHERE ar1.MemberId = {0} AND ar2.MemberId != {0}
            ) AS Candidates
        ", user1).ToListAsync();

        foreach (var user2 in potentialConnections.Take(20))
        {
            var similarity = await CalculateUserSimilarityAsync(user1, user2);
            trainingData.Add(new FriendData {
                UserId = user1,
                OtherUserId = user2,
                Label = Math.Max(0.01f, similarity)
            });
        }
    }

    return trainingData;
}

1 reference
private async Task SaveModelAsync()
{
    if (_model == null) return;
    var directory = Path.GetDirectoryName(_modelPath);
    if (!string.IsNullOrEmpty(directory) && !Directory.Exists(directory)) Directory.CreateDirectory(directory);
    await Task.Run(() => _mlContext.Model.Save(_model, null, _modelPath));
}

1 reference
private void LoadModelIfExists()
{
    try
    {
        if (File.Exists(_modelPath))
        {
            _model = _mlContext.Model.Load(_modelPath, out var modelInputSchema);
        }
    }
    catch (Exception ex)
```



```
    catch (Exception ex)
    {
        _logger.LogError(ex, "Error loading model from {ModelPath}", _modelPath);
    }
}

1 reference
private PredictionEngine<FriendData, FriendPrediction>? GetPredictionEngine()
{
    lock (_predictionEngineLock)
    {
        if (_cachedPredictionEngine == null || DateTime.Now - _lastPredictionEngineCreation > _predictionEngineLifetime)
        {
            _cachedPredictionEngine?.Dispose();
            _cachedPredictionEngine = _model != null
                ? _mlContext.Model.CreatePredictionEngine<FriendData, FriendPrediction>(_model)
                : null;
            _lastPredictionEngineCreation = DateTime.Now;
        }
        return _cachedPredictionEngine;
    }
}

1 reference
private (int iterations, int rank) GetOptimalTrainingParams()
{
    var userCount = _context.Members.Count();
    return userCount switch
    {
        < 100 => (30, 8),
        < 500 => (40, 16),
        < 2000 => (50, 32),
        < 10000 => (60, 64),
        _ => (80, 128)
    };
}

2 references
public async Task RetrainModelAsync()
{
    lock (_predictionEngineLock)
    {
        _model = null;
        _cachedPredictionEngine = null;
    }
    await TrainModelAsync();
}

```

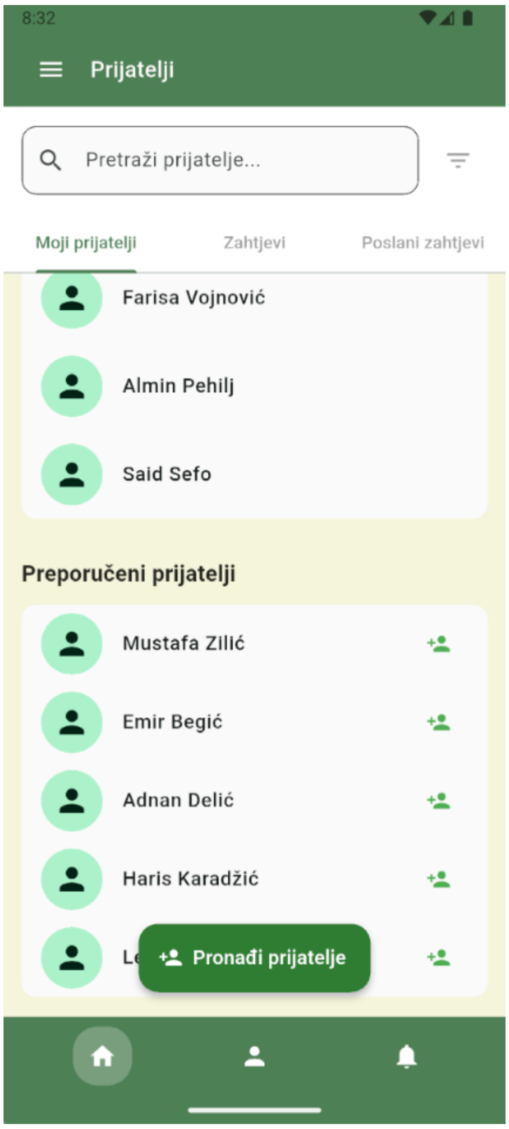


```
public void ClearRecommendationCache(int? userId = null)
{
    if (userId.HasValue)
    {
        _cache.Remove(userId.Value);
    }
    else
    {
        _cache.Clear();
    }
}

1 reference
public async Task WarmUpCacheAsync(int maxUsers = 50)
{
    try
    {
        var activeUsers = await _context.Members
            .Select(m => new { m.Id, ActivityScore = (m.ActivityRegistrations.Count(ar => ar.Status == RegistrationStatus.Approved)) * 2.0f +
                m.Posts.Count() * 1.5f + m.Comments.Count() * 1.0f + m.Likes.Count() * 0.5f + m.Reviews.Count() * 1.2f })
            .OrderByDescending(x => x.ActivityScore).Take(maxUsers).Select(x => x.Id).ToListAsync();

        var warmUpTasks = activeUsers.Select(async userId =>
        {
            try { await RecommendFriendsInternalAsync(userId, null, 5, CancellationToken.None); }
            catch (Exception ex) { _logger.LogWarning(ex, "Failed to warm up cache for user {UserId}", userId); }
        });

        await Task.WhenAll(warmUpTasks);
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Error during cache warm-up");
    }
}

0 references
public object GetCacheStatistics()
{
    var now = DateTime.UtcNow;
    var expiredCount = _cache.Count(kvp => now - kvp.Value.Timestamp > _cacheExpiry);
    return new
    {
        TotalCachedUsers = _cache.Count,
        ActiveCacheEntries = _cache.Count - expiredCount,
        ExpiredCacheEntries = expiredCount,
        CacheExpiryMinutes = _cacheExpiry.TotalMinutes
    };
```

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

ScoutTrack\ScoutTrack\ScoutTrack.UI\scouttrack_mobile\lib\screens\friendship_screen.dart

Printscreen iz pokrenute aplikacije:



Farisa Vojnović

Almin Pehilj

Said Sefo

Preporučeni prijatelji

Mustafa Zilić

Emir Begić

Adnan Delić

Haris Karadžić

Le