Perceptron

Feature	Our code(Scikit-Learn)	Ekaterina's Code (PyTorch)
Framework	Scikit-Learn	PyTorch
model	Perceptron from sklearn.linear_model	Neural network with nn.Linear
optimization	SGD with Perceptron default optimizer	Adam optimizer
Loss function	Hinge loss (default for Perceptron)	Cross-Entropy loss
Training process	One-step fitting	Mini-batch training with DataLoader
Weight extraction	No weight analysis	Returns absolute model weights

What Ekaterina’s Code Does

1. **Data Preparation**
   - Converts `X_train` and `y_train` into PyTorch tensors.
   - Uses `TensorDataset` and `DataLoader` for efficient batch processing.
2. **Model Definition**
   - Uses a simple neural network (`nn.Linear`) with one output layer and softmax activation.
   - Output has two neurons (one for each class: Feminine vs. Masculine).
3. **Training Loop**
   - Runs for 20 epochs.
   - Uses Adam optimizer and CrossEntropyLoss.
   - Performs mini-batch training (batch size = 32).
4. **Feature Importance Extraction**
   - Extracts absolute values of the second row of model weights (`abs(model[0].weight.data[1].numpy())`).

What Our Code Does

Data Preparation

- Reads multiple datasets, ensuring each contains "Gender" labels and word embeddings.
- Drops unnecessary columns ("Word") to keep only embeddings and labels.
- Selects a subset of the dataset (`dataset_percentage = 99%`) to control training size.
- Splits data into training (80%) and testing (20%) while maintaining class balance.
- Standardizes embeddings using `StandardScaler` for better model convergence.

Model Definition

- Uses Scikit-Learn's `Perceptron` classifier for gender classification.
- The perceptron is a linear classifier that updates weights using the SGD algorithm (Stochastic Gradient Descent).
- Unlike Ekaterina's neural network, our perceptron does not use softmax or probability-based outputs.

Training & Evaluation

- Trains the perceptron using 1000 iterations (or until convergence).
- Uses binary labels (0 = Feminine, 1 = Masculine).
- Evaluates model performance using accuracy and `classification_report`.
- Stores trained models and test data for later analysis.

Performance Analysis

- Compares gender classification accuracy across different embedding models (FlauBERT, CamemBERT, XLM-R, mBERT, DistilBERT).
- Finds that FlauBERT-Small outperforms FlauBERT-Large, which contrasts with Ekaterina's results.

Why Does FlauBERT-Small Perform Better than FlauBERT-Large Here?

Ekaterina's model is different from Scikit-Learn's perceptron, so it's possible that:

1. **Training Strategy**: Adam optimizer adapts better to small models.
2. **Feature Representation**: Smaller models like FlauBERT-Small might generalize better for gender classification.
3. **Batch Training Effect**: The model updates weights differently from Scikit-Learn's perceptron

Procedure for Applying SHAP and Retraining Perceptron

1. Compute SHAP Feature Importance

- For each trained perceptron model, retrieve the stored test dataset.
- Initialize a SHAP Explainer (`shap.Explainer`) using the perceptron's prediction function.
- Generate SHAP values for all features, limiting evaluations to `2 × embedding_dim + 1`.
- Compute the mean absolute SHAP values for each feature and Store feature importance values for further analysis.
- Visualize important features using `shap.summary_plot()`.

Python Package:

- `shap` (SHapley Additive exPlanations)
- `numpy` (for numerical operations)

2. Store SHAP Feature Importance Results

- Normalize feature importance values across all models. And Store feature importance values in a structured DataFrame.
- We Rank features based on their average importance across models and Save the results to a CSV file.

3. Select Top Features Using SHAP

- Define feature selection thresholds (10%, 20%, 30%).and Sort features in descending order based on SHAP importance.
- We Extract the top-ranked features according to the defined thresholds and Store selected feature subsets for each model.

4. Retrain Perceptron Using SHAP-Selected Features

- Load the dataset and filter out non-feature columns.
- Subset the dataset to retain only the top SHAP-selected features.
- Train a new perceptron model (`sklearn.linear_model.Perceptron`) using only these features. Then we Evaluate accuracy on a held-out test set.

5. Compare SHAP-Selected Features vs. Full Features

- Extract baseline accuracy from full-feature perceptron models.
- Train perceptron models with SHAP-selected features at different selection thresholds.
- Compare classification performance between full-feature and SHAP-reduced models and Visualize performance trends using bar plots.

Applying LIME to Identify Important Features

Procedure:

1. Define dataset percentages for word selection. In this case, we analyze 1% of the dataset.
2. For each trained model, extract feature embeddings while maintaining class balance.

3. Initialize LIME explainer (`LimeTabularExplainer`) with:

- Feature names (`Dim i` for each embedding dimension).
- Class labels (`Masculine`, `Feminine`).
- Mode = "classification" (as we are working with binary classification).

3. Define a probability-like function to adapt the Perceptron's decision function for LIME.
4. Iterate over a sampled subset of words, applying LIME to generate feature importance scores.
5. Aggregate and normalize importance values across instances.

Results:

- LIME feature importance values were computed for each trained model and stored for further analysis.

Selecting Top Features and Retraining Perceptron

Following LIME-based feature selection, we retrain the perceptron model using only the most relevant features to assess whether dimensionality reduction improves classification performance.

Procedure:

1. Define user-specified feature selection percentages (10%, 30%, 40%).
2. Extract the LIME importance scores from the stored results.
3. Rank features based on their contribution to the model's predictions.
4. Select the top-ranked features according to the defined percentages.
5. Retrain a perceptron model using only these features.
6. Evaluate accuracy on a held-out test set.