“Practical Machine Learning” Competition –Visual Sentence Complexity

Toma Sabin Sebastian – Group 412

This project aims to predict scores based on textual data using machine learning models. The workflow includes extensive preprocessing, feature extraction through word processing methods and testing multiple regression models. A comprehensive hyperparameter tuning strategy is employed to optimize model performance.

In my research, I used the following machine learning algorithms: XGBoost, LightGBM, Ridge, Random Forest, K-NN, and SVM. Each model is evaluated for its predictive accuracy and generalization capability., along with natural language processing methods, TF-IDF and Word2Vec to represent text in numerical form, enabling the machine learning models to process it.

The Spearman coefficient recorded significant changes when I switched the word processing method from TF-IDF to Word2Vec. My first model was Ridge, but later in my research, I discovered that XGBoost would be more suitable for this task. The same happened with natural language processing methods because on my first tries I utilized TF-IDF and later, it was changed by Word2Vec. I used text preprocessing methods such as removing stopwords, non-alphabetic characters, and converting text to lowercase. I conducted various experiments with Random Forest, K-NN, and SVM, adding these algorithms alongside XGBoost for model training. However, the model that showed the most improvement was LightGBM. To determine the optimal parameters, I used GridSearch, tweaking them over several attempts to find the best ones. This involves iteratively adjusting parameters such as the regularization strength, tree depth, learning rate, and others, to identify the optimal configuration. Special attention is given to evaluating the models using metrics like Spearman’s Rank Correlation Coefficient, Mean Absolute Error Mean Squared Error and Kendall’s Tau Correlation Coefficient on validation set to ensure robust and reliable predictions. Additionally, experiments with data standardization were conducted, but these approaches yielded suboptimal results, emphasizing the need for tailored preprocessing techniques specific to textual data. The final model leverages the LightGBM algorithm, which demonstrated superior performance in terms of predictive accuracy and computational efficiency, making it an excellent choice for this task.

The 2 submissions I want to talk about are:

1. Ridge & TF-IDF
2. XGBoost, LightBM, Word2Vec & GridSearch

1. Ridge & TF-IDF submission

Ridge regression, also known as Tikhonov regularization, is a technique used in linear regression to address the problem of multicollinearity among predictor variables. Multicollinearity occurs when independent variables in a regression model are highly correlated, which can lead to unreliable and unstable estimates of regression coefficients. Ridge regression is a procedure for eliminating the bias of coefficients and reducing the mean square error by shrinking the coefficients of a model towards zero in order to solve problems of overfitting or multicollinearity that are normally associated with ordinary least squares regression. **TF-IDF** stands for Term Frequency Inverse Document Frequency of records. It can be defined as the calculation of how relevant a word in a series or corpus is to a text. The meaning increases proportionally to the number of times in the text a word appears but is compensated by the word frequency in the corpus (data-set).

The code designed to build and evaluate a text-based machine learning regression model for predicting scores. It evaluates Ridge regression as the model and applies TF-IDF vectorization for text feature extraction.

- For building this model, I used libraries such as: Pandas and NumPy for data manipulation, SciPy and Scikit-learn for statistical analysis and machine learning. NLTK for natural language preprocessing, specifically for removing stopwords. Regular Expressions (re) for text cleaning.

- Text Preprocessing Techniques: Removing non-alphabetic characters: Using regular expressions to retain only letters and spaces. Converting to lowercase: Ensures uniformity across text data. Tokenization and stopword removal: Each text is split into individual words, and common words (stopwords) are filtered out using the NLTK stopword list. This ensures that the textual data is clean and focuses on meaningful content.

```
stop_words = set(stopwords.words('english'))

# Functia de preprocesare a textului
def preprocess_text(text):  3 usages
    text = re.sub( pattern: r'[^a-zA-Z\s]',  repl: '', text.lower())  # Eliminăm caracterele non-alfabetice
    tokens = text.split()  # Împărtim textul în cuvinte
    tokens = [word for word in tokens if word not in stop_words]  # Eliminăm stop word-urile
    return ' '.join(tokens)
```

Figure 1. Text Prepocessing Techniques

```
# Curatarea textului
train["text"] = train["text"].apply(preprocess_text)
val["text"] = val["text"].apply(preprocess_text)
test["text"] = test["text"].apply(preprocess_text)
```

Figure 2. Applying the preprocessing techniques

- Adding train, validation and test set: These lines ensure that the data is ready for preprocessing, feature extraction, model training, and evaluation. The training dataset typically contains input data and corresponding target labels or scores that are used to train the machine learning model. The test dataset contains input data for which predictions need to be made. The validation dataset is used to evaluate the model's performance during the training process. It helps in tuning hyperparameters and avoiding overfitting by providing an unbiased evaluation.

```
# Incarcarea datelor
train = pd.read_csv('Date/train.csv')
test = pd.read_csv('Date/test.csv')
val = pd.read_csv('Date/val.csv')
```

Figure 3. Adding data

- Data Augmentation: For the data augmentation process, I introduced noise by adding a small amount of random noise to the target scores in the training set. This step introduces variability and helps prevent the model from overfitting to specific patterns in the training data.

```
# Adaugarea unui zgomot aleator scorurilor tinta
noise_level = 0.01
train["score"] += np.random.normal( loc: 0, noise_level, size=len(train))
```

Figure 4. Adding noise

- Text Vectorization: I continued with text vectorization using TF-IDF with the following parameters: **Maximum features**: Limits the number of terms considered. **N-grams**: Captures both single words (unigrams) and consecutive word pairs (bigrams). **Minimum document frequency**: Filters out rare terms that appear in fewer than two documents.

```
# Vectorizarea textului
vectorizer = TfidfVectorizer(max_features=30000, ngram_range=(1, 3), min_df=2)
x_train = vectorizer.fit_transform(train["text"])
x_val = vectorizer.transform(val["text"])
x_test = vectorizer.transform(test["text"])
```

Figure 5. Vectorization

- Model Training: The Ridge regression model was trained on the vectorized text data and the corresponding scores. Ridge regression was chosen for its ability to handle high-dimensional data effectively and reduce overfitting.

```
# Scorurile tinta
y_train = train["score"]
y_val = val["score"]

# Antrenarea modelului
model = Ridge(alpha=1)
model.fit(x_train, y_train)
```

Figure 6. Training

- **Evaluation Metrics:** The model's performance was evaluated using the following metrics: **Spearman's Rank Correlation Coefficient:** Measures the monotonic relationship between the true and predicted scores. **Mean Absolute Error (MAE):** Captures the average magnitude of prediction errors. **Mean Squared Error (MSE):** Highlights larger errors due to its quadratic nature. **Kendall's Tau Correlation Coefficient:** Provides another perspective on the rank-based relationship.

```
# Predictii
val_predictions = model.predict(x_val)
spearman_corr, _ = spearmanr(y_val, val_predictions)
print(f"Spearman's Rank Correlation Coefficient on validation set: {spearman_corr:.4f}")
```

Figure 7. Spearman coefficient in validation dataset

```python
from sklearn.metrics import mean_absolute_error, mean_squared_error
from scipy.stats import kendalltau

mae = mean_absolute_error(y_val, val_predictions)
mse = mean_squared_error(y_val, val_predictions)
kendall_corr, _ = kendalltau(y_val, val_predictions)

print(f"Mean Absolute Error (MAE) on validation set: {mae:.4f}")
print(f"Mean Squared Error (MSE) on validation set: {mse:.4f}")
print(f"Kendall's Tau Correlation Coefficient on validation set: {kendall_corr:.4f}")
```

Figure 8. MAE, MSE & Kendall' Tau coefficient

- Predictions and Output:Predictions were generated and saved to a CSV file, containing two columns**:** ID and Score.

```python
# Predictii pe setul de test
test_predictions = model.predict(x_test)

# Salvarea rezultatelor
submission = pd.DataFrame({
    "id": test["id"],
    "score": test_predictions
})
submission.to_csv( path_or_buf: "DateOUT/Submission.csv", index=False)
```

Figure 9. Saving prediction on test dataset

Below are the inputs used throughout the attempts for the Ridge & TF-IDF model. The order was random, and I aimed for each run to differ by at least one value from the others.

Noise	TF-IDF Vectorizer
0.01	vectorizer = TfidfVectorizer(max_features=10000, ngram_range=(1, 2), min_df=2)
0.1	vectorizer = TfidfVectorizer(max_features=20000, ngram_range=(1, 2), min_df=2)
0.03	vectorizer = TfidfVectorizer(max_features=20000, ngram_range=(1, 3), min_df=2)
0.5	vectorizer = TfidfVectorizer(max_features=10000, ngram_range=(1, 3), min_df=2)
0.7	vectorizer = TfidfVectorizer(max_features=30000, ngram_range=(1, 3), min_df=2)
0.23	vectorizer = TfidfVectorizer(max_features=15000, ngram_range=(1, 4), min_df=2)
0.19	vectorizer = TfidfVectorizer(max_features=15000, ngram_range=(1, 2), min_df=3)
0.02	vectorizer = TfidfVectorizer(max_features=50000, ngram_range=(1, 2), min_df=2)
	vectorizer = TfidfVectorizer(max_features=100000, ngram_range=(1, 2), min_df=2)
	vectorizer = TfidfVectorizer(max_features=100000, ngram_range=(1, 3), min_df=4)
	vectorizer = TfidfVectorizer(max_features=100000, ngram_range=(1, 3), min_df=2)
	vectorizer = TfidfVectorizer(max_features=100000, ngram_range=(1, 5), min_df=1)
	vectorizer = TfidfVectorizer(max_features=100000, ngram_range=(1, 5), min_df=2)
	vectorizer = TfidfVectorizer(max_features=100000, ngram_range=(1, 10), min_df=1)

Table 1. Noise values and TF-IDF embeddings

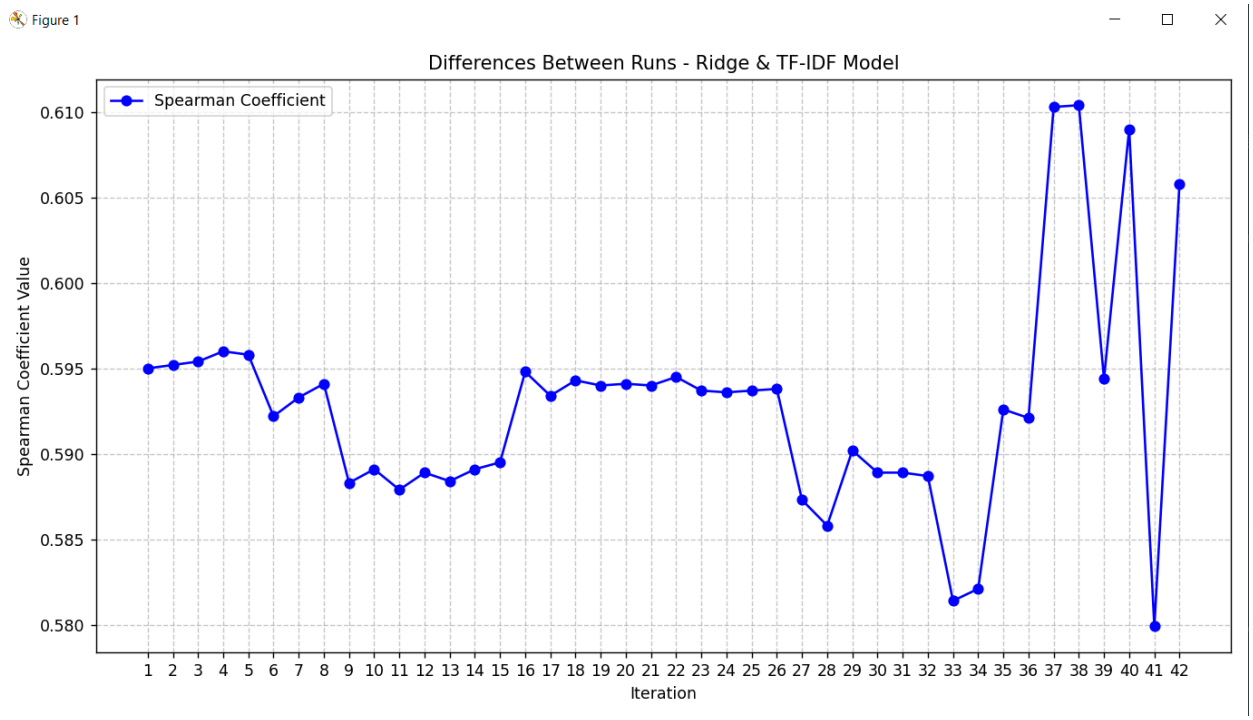

Figure 10. Spearman's coefficient first model

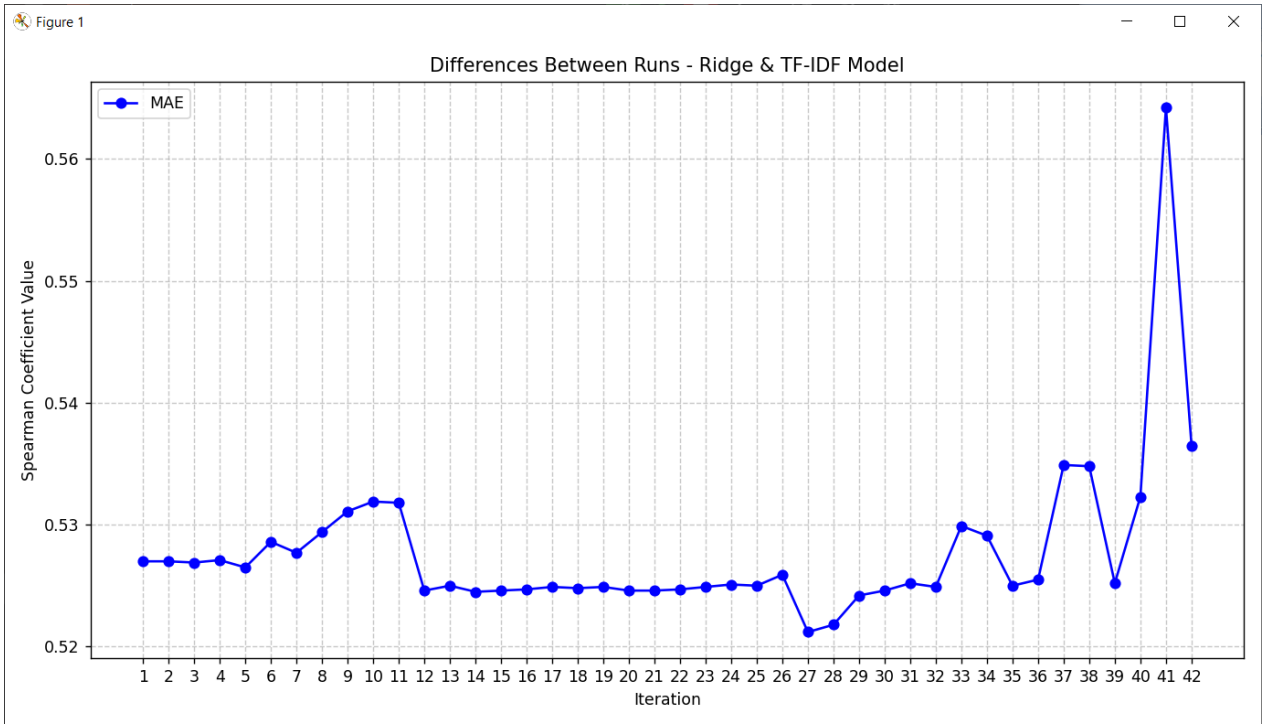

Figure 11. MAE first model

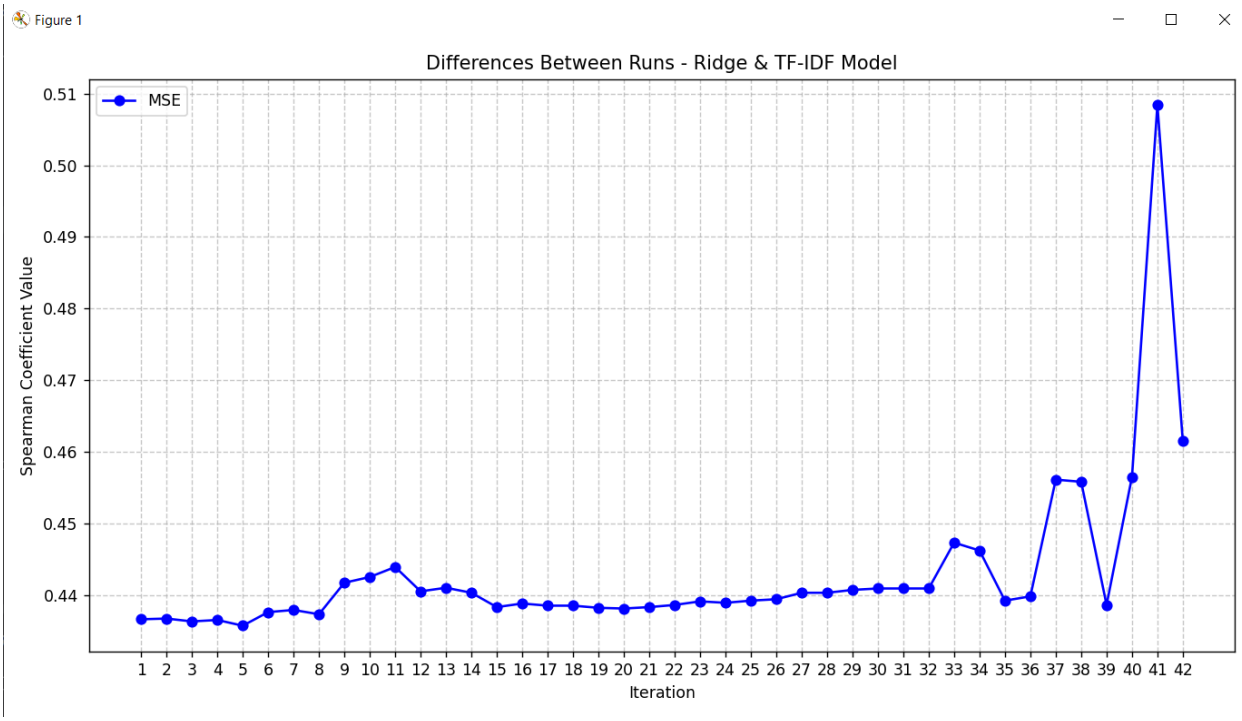

Figure 12. MSE first model

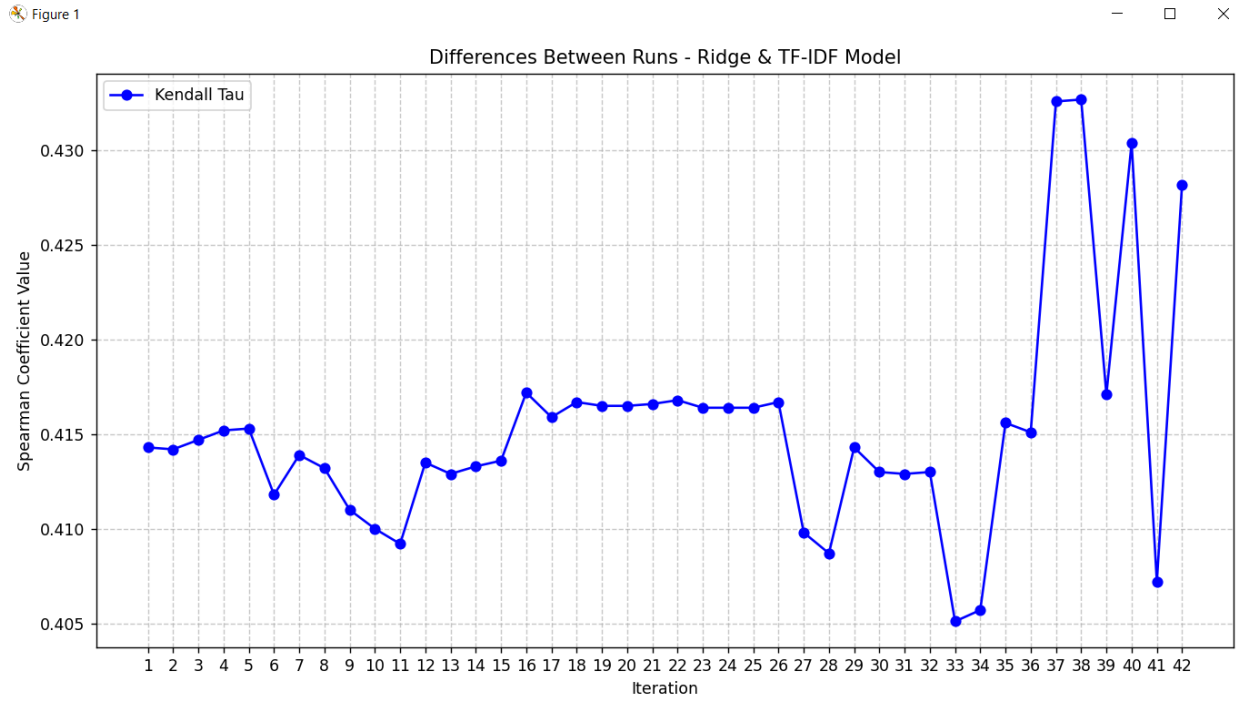

Figure 13. Kendall's Tau first model

vectorizer = TfidfVectorizer(max_features=100000, ngram_range=(1, 10), min_df=1)

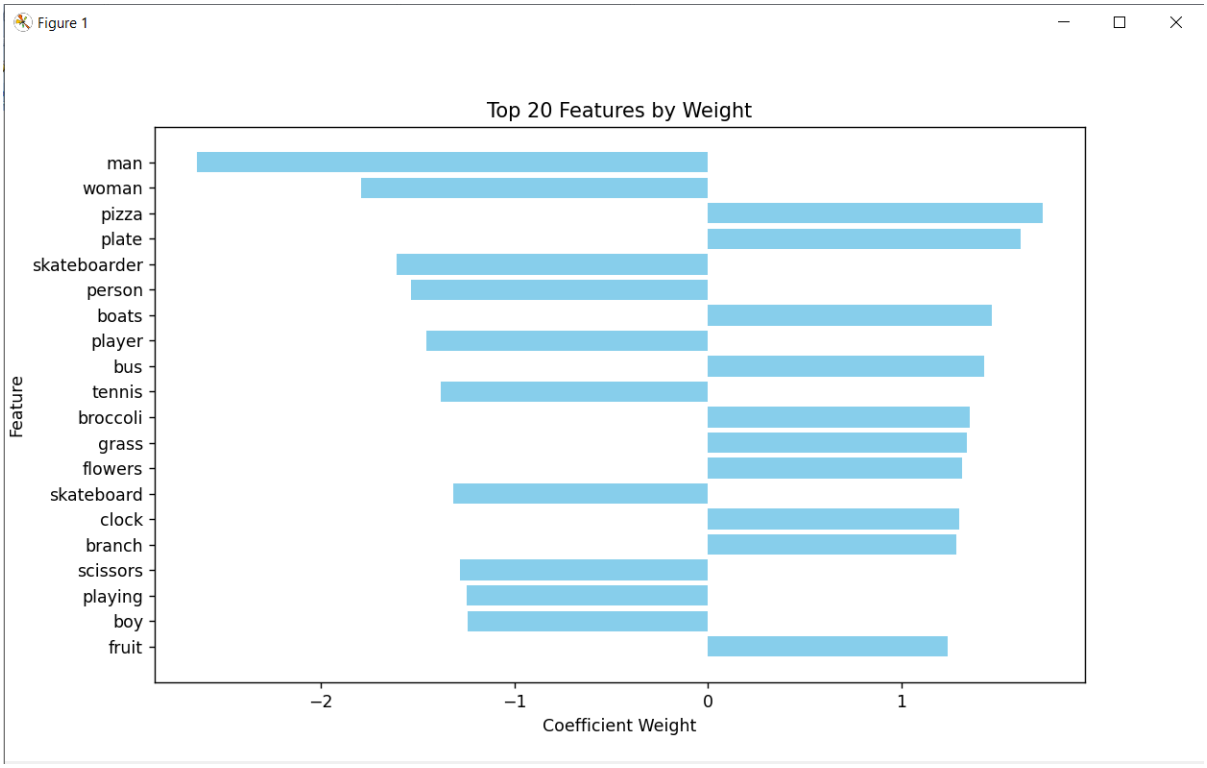

Figure 14. Top 20 features by their coefficient weights in Ridge regression

1. Features: The y-axis lists the top 20 features (words or terms) from the TF-IDF vectorization process. These are the most influential terms for the model when predicting the target score.

2. Coefficient Weight: The x-axis represents the Ridge regression model's coefficients for each feature. The coefficients indicate how much a given feature impacts the prediction:

- Positive weights (right of 0): These features increase the predicted score. For example, terms like "pizza" and "plate" have high positive weights.
- Negative weights (left of 0): These features decrease the predicted score. For instance, terms like "man" and "woman" have large negative weights.

3. Insights:

- Features like "pizza," "plate," and "skateboarder" are positively associated with the target scores, meaning they contribute more to higher scores.
- Features like "man" and "woman" are negatively associated, meaning their presence reduces the predicted score.
- The magnitude of the coefficient indicates the feature's importance. Larger absolute values have a stronger influence on the model's predictions.

vectorizer = TfidfVectorizer(max_features=50000, ngram_range=(1, 2), min_df=2)

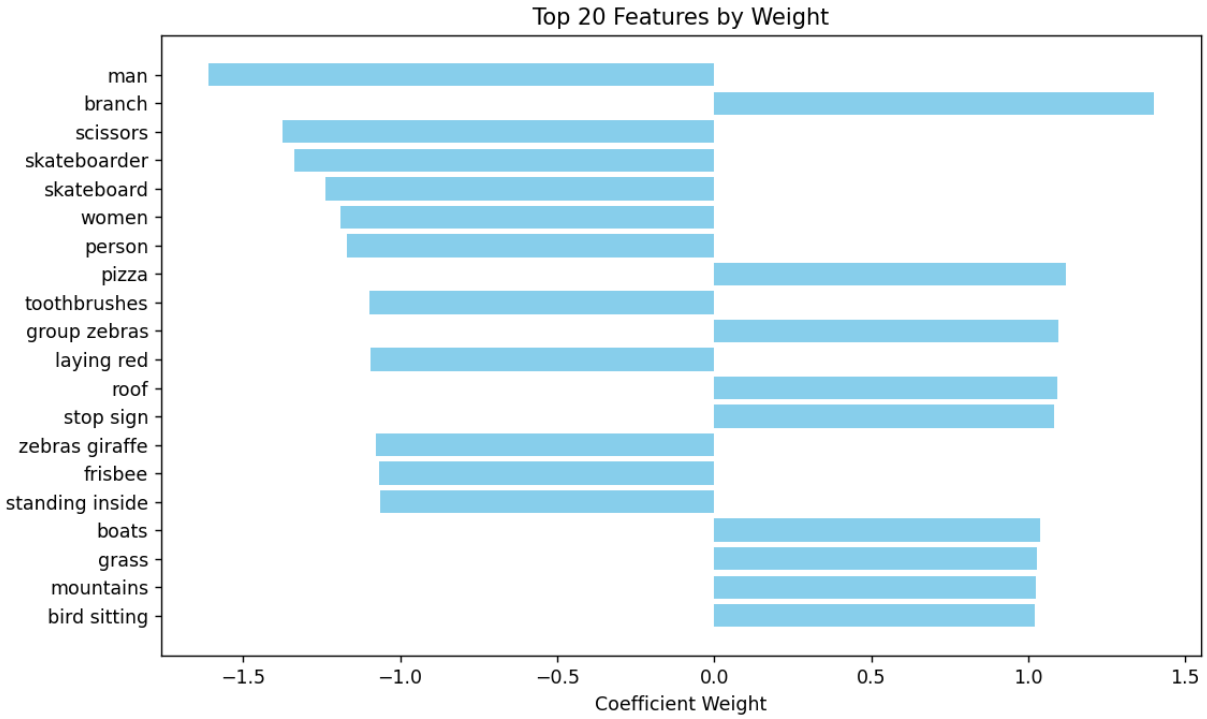

Figure 15. Top 20 features by their coefficient weights in Ridge regression

vectorizer = TfidfVectorizer(max_features=10000, ngram_range=(1, 3), min_df=2)

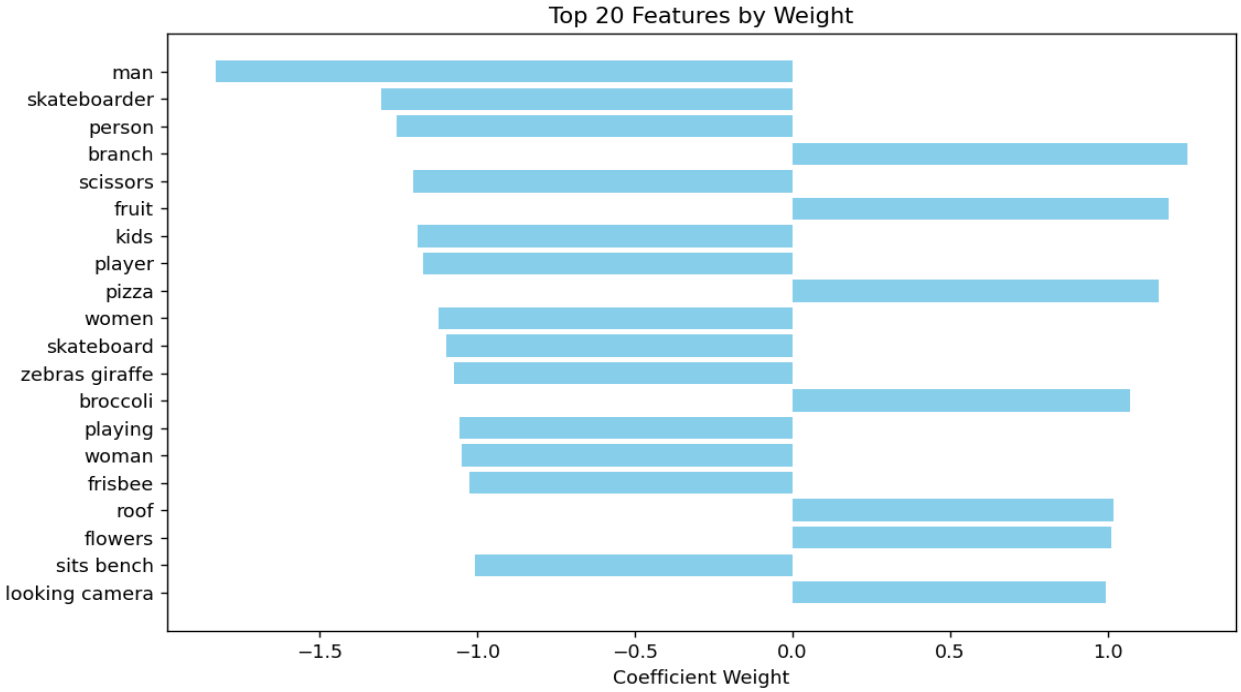

Figure 16. Top 20 features by their coefficient weights in Ridge regression

2. XGBoost, LightBM, Word2Vec & GridSearch

XGBoost(Extreme Gradient Boosting) is an optimized distributed gradient boosting library designed for efficient and scalable training of machine learning models. It is an ensemble learning method that combines the predictions of multiple weak models to produce a stronger prediction. One of the key features of XGBoost is its efficient handling of missing values, which allows it to handle real-world data with missing values without requiring significant pre-processing. Additionally, XGBoost has built-in support for parallel processing, making it possible to train models on large datasets in a reasonable amount of time. **LightGBM** (Light Gradient Boosting Machine) is an ensemble learning framework, specifically a gradient boosting method, which constructs a strong learner by sequentially adding weak learners in a gradient descent manner. It optimizes memory usage and training time with techniques like Gradient-based One-Side Sampling (GOSS). Additionally, LightGBM employs histogram-based algorithms for efficient tree construction.

Word2Vec is a widely used method in natural language processing (NLP) that allows words to be represented as vectors in a continuous vector space. Word2Vec is an effort to map words to high-dimensional vectors to capture the semantic relationships between words, developed by researchers at Google. Words with similar meanings should have similar vector representations, according to the main principle of Word2Vec. Word2Vec utilizes two architectures: CBOW (Continuous Bag of Words) and Skip Gram. A Machine Learning model is defined as a mathematical model with a number of parameters that need to be learned from the data. However, there are some parameters, known as **Hyperparameters** and those cannot be directly learned. They are commonly chosen by humans based on some intuition or hit and trial before the actual training begins. These parameters exhibit their importance by improving the performance of the model such as its complexity or its learning rate. Models can have many hyper-parameters and finding the best combination of parameters can be treated as a search problem. **GridSearchCV** takes a dictionary that describes the parameters that could be tried on a model to train it. The grid of parameters is defined as a dictionary, where the keys are the parameters and the values are the settings to be tested.

This code demonstrates a robust machine learning pipeline to predict scores based on text data. Key aspects include text preprocessing, feature extraction with Word2Vec, and advanced regressors (XGBoost and LightGBM) with hyperparameter optimization.

- In addition to libraries such as Pandas, NumPy, NLTK, Re, Scipy, and Scikit-learn, which are also used in the Ridge-TFIDF model, gensim was used for Word2Vec, a word processing method, XGBRegressor for XGBoost, and LGBMRegressor for LightGBM. After that, it was written the data loading: train, test and validations dataset and the same text preprocessing methods that we were used in the previous model.

- Word Embedding with Word2Vec that has the purpose to convert text into numerical vectors that capture semantic meaning having following parameters. **Sentences** is the list of propositions, every proposition being a list of words/tokens where Word2Vec trains. **Vector_size** is the dimensionality of the vectors representing each word. **Window** defines how many neighboring words, both to the left and right of a central word, are considered during training**. Min_count** is the minimum frequency threshold for a word to be included in the dataset where words that occur less frequently than the specified value are ignored during training. **Workers** are the number of CPU cores used during training. **Sg** determines the training algorithm used. Sg=1 results into Skip-gram which trains to predict surrounding words given a central word. Sg=0 is CBOW (Continuous Bag of Words) that predicts the central word based on its surrounding context. Epochs is the number of complete passes through the dataset during training.

```
# Antrenarea unui model Word2Vec pe textul tokenizat
all_tokens = (train["tokens"].tolist() + val["tokens"].tolist() + test["tokens"].tolist())
word2vec_model = Word2Vec(sentences=all_tokens, vector_size=150, window=4, min_count=1, workers=5, sg=1, epochs=50)
```

Figure 17. Word2Vec hyperparameters

```
# Functie pentru a calcula vectorul mediu pentru fiecare propozitie
def compute_sentence_vector(tokens, model, vector_size):  3 usages
    vectors = [model.wv[word] for word in tokens if word in model.wv]
    if len(vectors) > 0:
        return np.mean(vectors, axis=0)
    else:
        return np.zeros(vector_size)
```

Figure 18. Function for finding central word for every proposition

```
# Calcularea vectorilor pentru seturile de date
vector_size = word2vec_model.vector_size
x_train = np.array([compute_sentence_vector(tokens, word2vec_model, vector_size) for tokens in train["tokens"]])
x_val = np.array([compute_sentence_vector(tokens, word2vec_model, vector_size) for tokens in val["tokens"]])
x_test = np.array([compute_sentence_vector(tokens, word2vec_model, vector_size) for tokens in test["tokens"]])

# Scorurile țintă
y_train = train["score"]
y_val = val["score"]
```

Figure 19. Text vectorization

- Regression models: I created a custom class that wraps two popular regression models: XGBoost(Optimized distributed gradient boosting library) and

LightGBM(gradient boosting method) that Supports model selection via a *model_type* parameter and allows dynamic parameter updates for XGBoost and LightGBM.

```python
# Definirea unui estimatori personalizati pentru a integra XGBoost și LightGBM
class MultiModelRegressor(BaseEstimator, RegressorMixin):  1 usage
    def __init__(self, model_type="xgboost", **params):
        self.model_type = model_type
        self.params = params
        self.model = None

    def set_params(self, **params):
        """Updates model-specific parameters."""
        if 'model_type' in params:
            self.model_type = params.pop('model_type')
        self.params.update(params)
        return self

    def fit(self, X, y):  1 usage (1 dynamic)
        if self.model_type == "xgboost":
            self.model = XGBRegressor(**self.params, random_state=42)
        elif self.model_type == "lightgbm":
            self.model = LGBMRegressor(**self.params, random_state=42)
        else:
            raise ValueError("Unsupported model_type. Use 'xgboost' or 'lightgbm'."
        self.model.fit(X, y)
        return self

    def predict(self, X):  10 usages (10 dynamic)
        return self.model.predict(X)
```

Figure 20. Custom class for choosing the best options

- Hyperparameter tuning: GridSearchCV finds the best combination of hyperparameters for each model. For XGBoost:n_estimators, learning_rate, max_depth, subsample, col_sample_bytree. For LightGBM: n_estimators, learning_rate, max_depth, num_leaves, min_child_samples, subsample. Pipeline setup includes a StandardScaler to normalize input features and uses the custom class to handle model tasks. Scoring metric: Negative mean squared error(MSE).

```
# Setul de hiperparametri pentru GridSearchCV
param_grid = [
    {

        'model__model_type': ['xgboost'],
        'model__n_estimators': [100, 200, 300, 400],
        'model__learning_rate': [0.01, 0.05, 0.04, 0.1, 0.08, 0.13],
        'model__max_depth': [5, 7, 10],
        'model__subsample': [0.8, 1.0],
        'model__colsample_bytree': [0.8, 1.0]
    },
```

Figure 21. XGBoost hyperparameters

```
{

    'model__model_type': ['lightgbm'],
    'model__n_estimators': [100, 200, 300, 400],
    'model__learning_rate': [0.01, 0.05, 0.1, 0.04, 0.08, 0.13],
    'model__max_depth': [5, 7, 10],
    'model__num_leaves': [31, 50, 70],
    'model__min_child_samples': [10, 20],
    'model__subsample': [0.8, 1.0],
}
```

Figure 22. LightGBM hyperparameters

```
# Definirea pipeline-ului
pipeline = Pipeline([
    ('scaler', StandardScaler()),  # Normalizarea datelor
    ('model', MultiModelRegressor())  # Alegerea modelului
])

# Configurarea GridSearchCV
grid_search = GridSearchCV(
    estimator=pipeline,
    param_grid=param_grid,
    scoring='neg_mean_squared_error',
    cv=3,
    verbose=2,
    n_jobs=-1
)
```

Figure 23. Pipeline and GridSearchCV configuration

Estimator=pipeline: Specify the base model that will be optimized. **Param_grid**=param_grid: A grid containing hyperparameters and their possible values**. Scoring**='neg_mean_squared_error': Specifies the metric used to evaluate the models. Neg_mean_squared_error is the negative mean square error. The negative value is used because the scores in scikit-learn are generally maximum (and not minimum). **Cv**=3:Number of folds for cross-validation. The data is divided into 3 folds: 2 folds for training and 1 for testing. This process is repeated for each combination of hyperparameters. **Verbose**=2:Controls the amount of information displayed during running: 0: No information, 1: General progress, 2: More detailed. **N_jobs**=-1:Determines the number of cores used for processing.-1 indicates the use of all available cores to speed up the process.

```
# Executarea Grid Search
grid_search.fit(x_train, y_train)

# Cei mai buni hiperparametri
best_params = grid_search.best_params_
print(f"Best parameters: {best_params}")

# Modelul optim
best_model = grid_search.best_estimator_

# Predictii pe setul de validare
val_predictions = best_model.predict(x_val)
spearman_corr, _ = spearmanr(y_val, val_predictions)
print(f"Spearman's Rank Correlation Coefficient on validation set: {spearman_corr:.4f}")
```

Figure 24. Finding the best hyperparameters and model

- As in the previous model, the Spearman, MAE, MSE and Kendall values were found on the validation set and the predictions in the test set.

```
from sklearn.metrics import mean_absolute_error, mean_squared_error
from scipy.stats import kendalltau

mae = mean_absolute_error(y_val, val_predictions)
mse = mean_squared_error(y_val, val_predictions)
kendall_corr, _ = kendalltau(y_val, val_predictions)

print(f"Mean Absolute Error (MAE) on validation set: {mae:.4f}")
print(f"Mean Squared Error (MSE) on validation set: {mse:.4f}")
print(f"Kendall's Tau Correlation Coefficient on validation set: {kendall_corr:.4f}")

# Predictii pe setul de testare
test_predictions = best_model.predict(x_test)

# Salvarea rezultatelor
submission = pd.DataFrame({
    "id": test["id"],
    "score": test_predictions
})
submission.to_csv( path_or_buf: "DateOUT/SubmissionModel2.csv", index=False)
```

Figure 25. MAE, MSE, Kendall and predictions on test dataset

```
Spearman's Rank Correlation Coefficient on validation set: 0.5875
Mean Absolute Error (MAE) on validation set: 0.5160
Mean Squared Error (MSE) on validation set: 0.4359
Kendall's Tau Correlation Coefficient on validation set: 0.4185
```

Figure 26. Results

- Best parameters: {'model__colsample_bytree': 0.8, 'model__learning_rate': 0.01, 'model__max_depth': 7, 'model__model_type': 'xgboost', 'model__n_estimators': 400, 'model__subsample': 0.8}

Word2Vec
word2vec_model = Word2Vec(sentences=all_tokens, vector_size=100, window=5, min_count=1, workers=4, sg=1, epochs=10)
word2vec_model = Word2Vec(sentences=all_tokens, vector_size=100, window=5, min_count=1, workers=4, sg=1, epochs=12)
word2vec_model = Word2Vec(sentences=all_tokens, vector_size=100, window=5, min_count=1, workers=4, sg=1, epochs=20)
word2vec_model = Word2Vec(sentences=all_tokens, vector_size=120, window=5, min_count=1, workers=5, sg=1, epochs=12)
word2vec_model = Word2Vec(sentences=all_tokens, vector_size=100, window=4, min_count=1, workers=2, sg=1, epochs=14)
word2vec_model = Word2Vec(sentences=all_tokens, vector_size=150, window=4, min_count=1, workers=5, sg=1, epochs=50)
word2vec_model = Word2Vec(sentences=all_tokens, vector_size=150, window=4, min_count=1, workers=5, sg=0, epochs=30)

Tabel 2. Word2Vec hyperparameters

XGBoost
'model__n_estimators': [100, 200, 300, 400]
'model__learning_rate': [0.01, 0.05, 0.04, 0.1, 0.08, 0.13]
'model__max_depth': [5, 7, 10]
'model__subsample': [0.8, 1.0]
'model__colsample_bytree': [0.8, 1.0]

Tabel 3. XGBoost hyperparameters

LightGBM
'model__n_estimators': [100, 200, 300, 400]
'model__learning_rate': [0.01, 0.05, 0.1, 0.04, 0.08, 0.13]
'model__max_depth': [5, 7, 10]
'model__num_leaves': [31, 50, 70]
'model__min_child_samples': [10, 20]
'model subsample': [0.8, 1.0]

Tabel 4. LightGBM hyperparameters

- To find out the results of this model, the parameters and hyperparameters from the tables were randomly used.

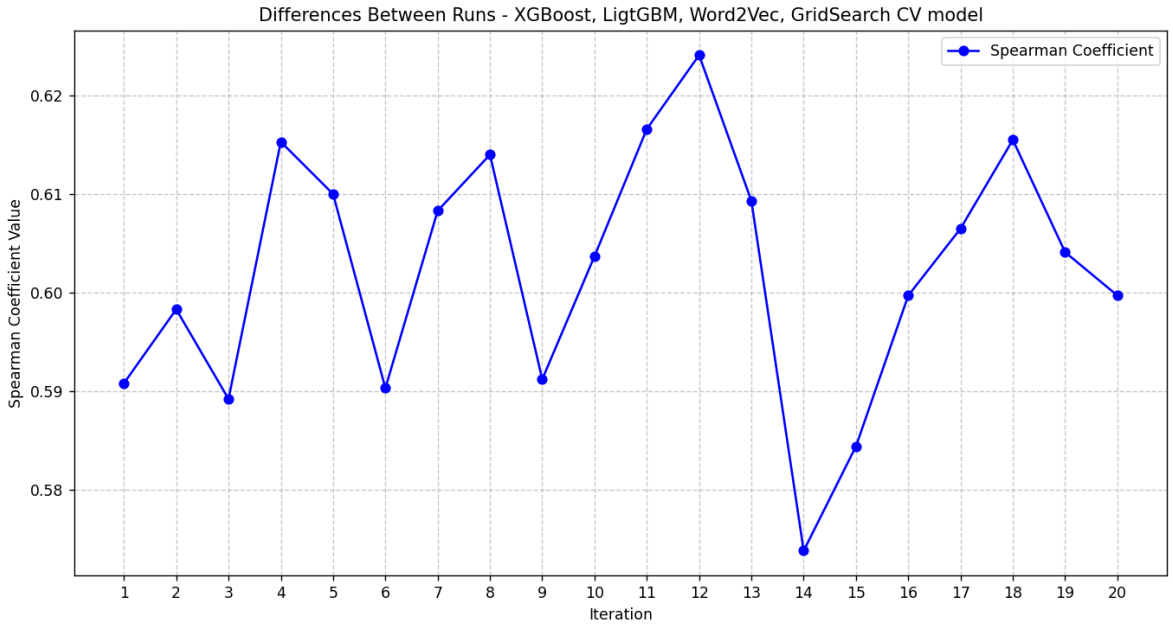

Figure 26. Spearman' s coefficient second model

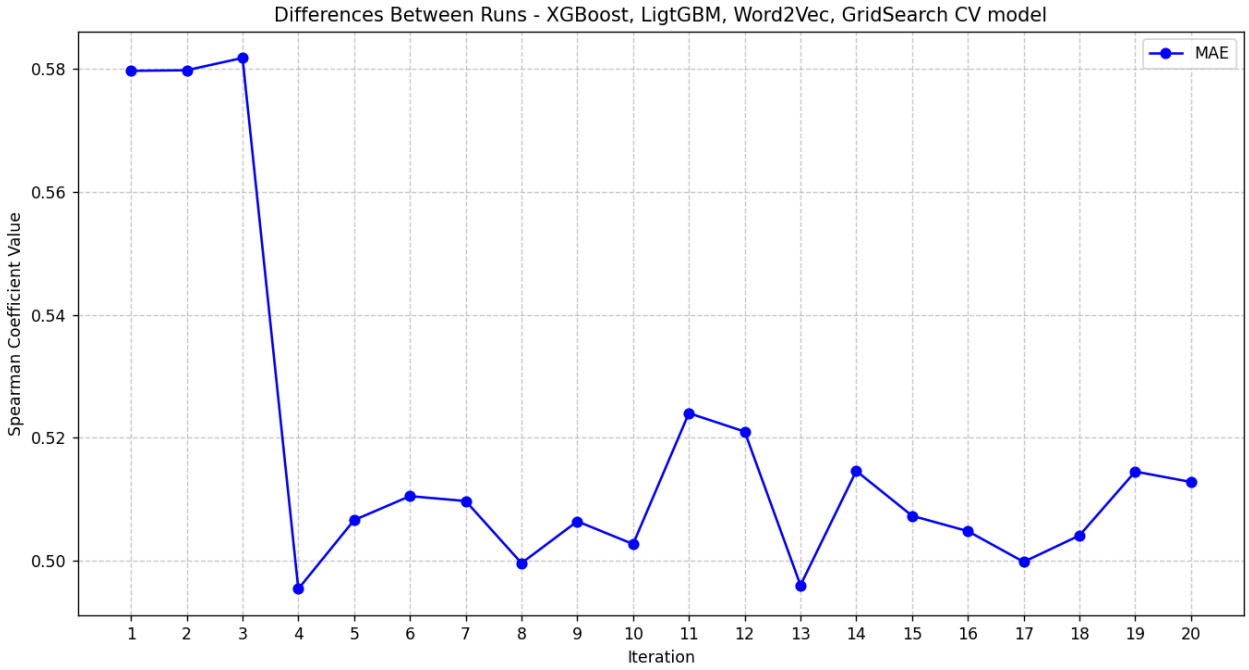

Figure 27. MAE second model

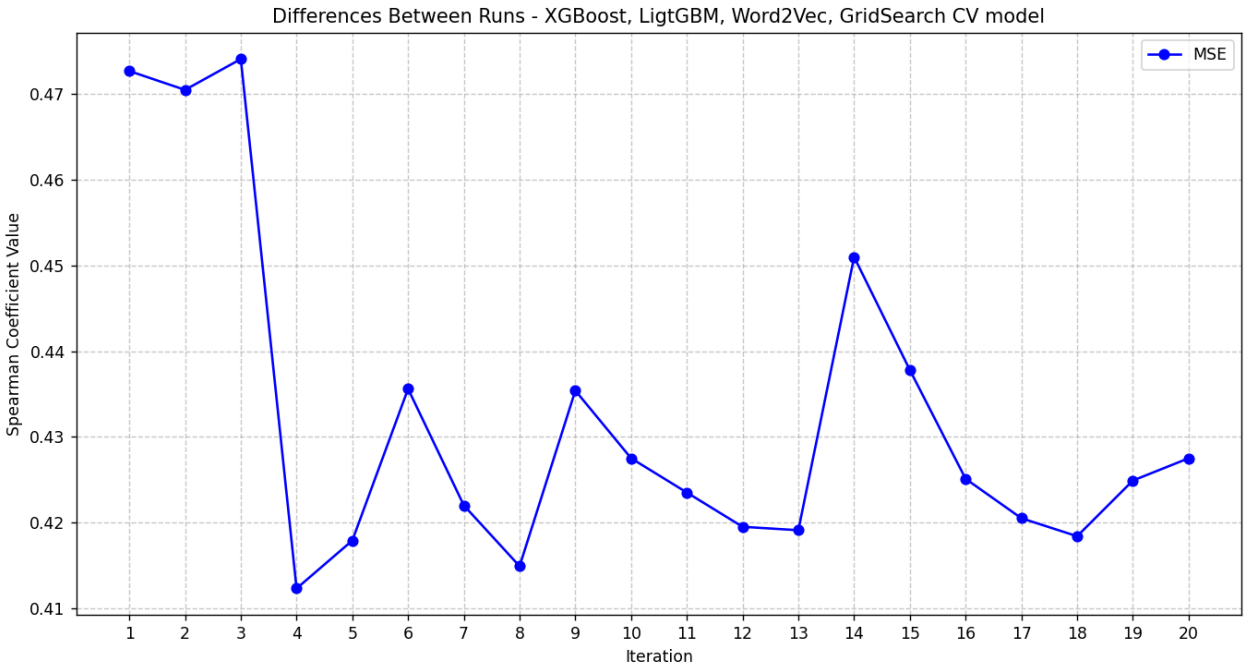

Figure 28. MSE second model

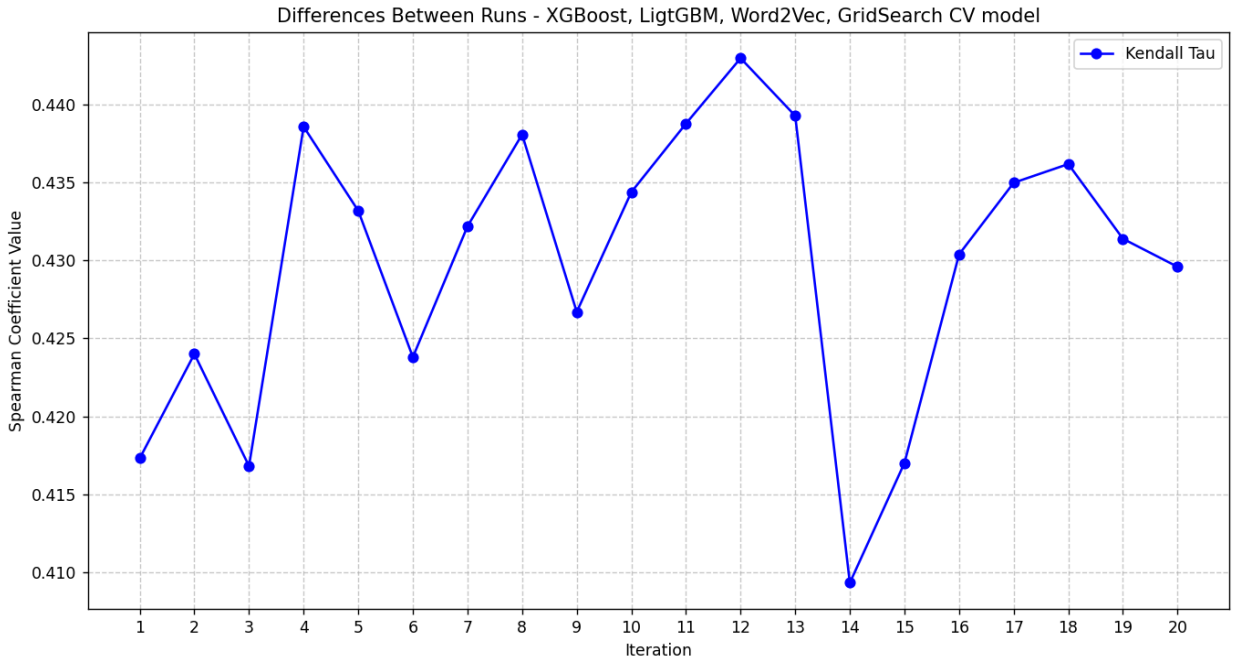

Figure 29. Kendall's Tau second model

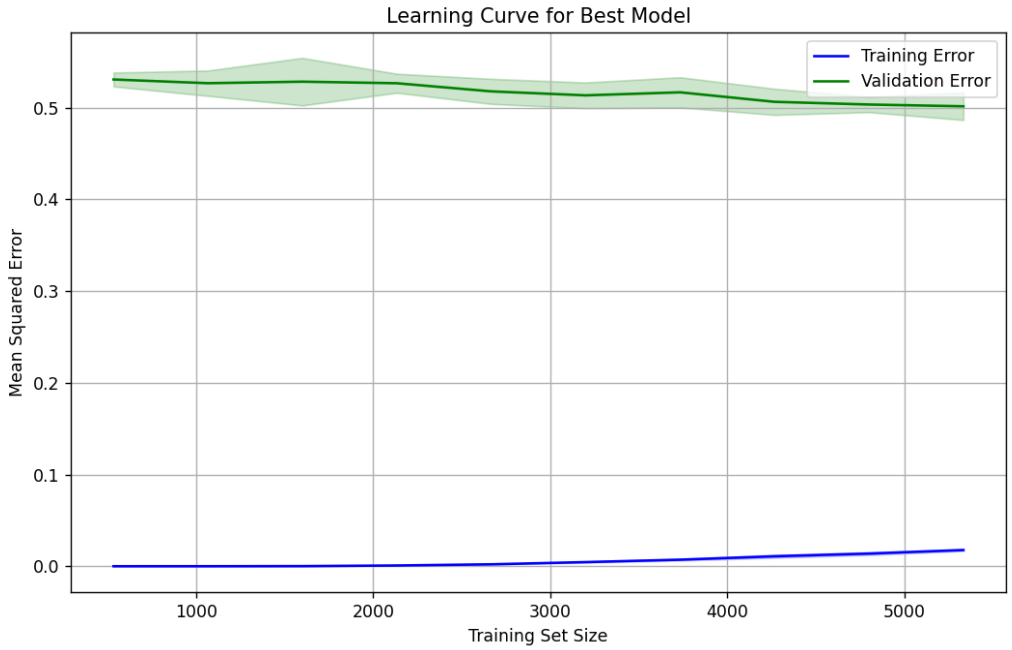

Figure 30. Learning Curve XGBoost, LightGBM, Word2Vec, GridSearchCV model

The graph presented above represents the learning process of the best model selected through the chosen hyperparameters.

On the X-axis is the training size, and on the Y-axis is the MSE, a metric for measuring errors between predicted and true values.

The blue line(Training error) shows the error of the "training" as the size of the training set increases. The green line (Validation error) sets the uncertainty range as the training set increases.

The trend of the blue line is to decrease as the training set increases. This means that more data will lead the model to learn better and be as efficient as possible. The trend of the green line starts high, but as the training set increases, it stabilizes.

The gap between the Training error and Validation error suggests that the dataset is large, which is a good indicator of a well-round model.

In this documentation we have presented one of the first models and one of the last models. The first model we formed was TF-IDF Ridge with no preprocessing, no noise. Then I continued with the one presented in point 1, and then Ridge with Word2Vec where there was a major change/improvement of the score from the competition, then I changed the model from XGBoost to Word2Vec. I managed to implement XGBoost along with GridSearchCV and other models, such as: K-NN, SVM,

Random Forest, but after some research, the best way to combine it was LightGBM, the second model presented.

In conclusion, this competition gave me the opportunity to analyze and implement regression models to make predictions that lead to accurate results by leveraging advanced text processing techniques, selecting the best model based on the task to be solved, and optimizing hyperparameters. Most of the models underwent an optimization process to generate scores on the test set, and the results were saved for further analysis.

Bibliography:

1. What is Ridge Regression? - GeeksforGeeks

2. Understanding TF-IDF (Term Frequency-Inverse Document Frequency) - GeeksforGeeks

3. ML | XGBoost (eXtreme Gradient Boosting) - GeeksforGeeks

4. LightGBM (Light Gradient Boosting Machine) - GeeksforGeeks

5. Word Embedding using Word2Vec - GeeksforGeeks

6. Comparing Randomized Search and Grid Search for Hyperparameter Estimation in Scikit Learn - GeeksforGeeks