Predicting Binding Affinity Using Pre-Computed Features

Group 08: P104, P303, P677, P995

March 24, 2025

Contents

1. Introduction

Our project goal is to identify the best model for predicting a molecule's binding affinity to a protein using pre-computed features. The dataset includes 1024 binary (fps) features, 224 real-valued (embed) features, and one real-valued target variable (binding affinity). It is divided into a training set (975 molecules), a test set (109 molecules), and a private test set (73 molecules, binding affinity unknown). We will analyze the dataset, apply necessary preprocessing, and evaluate various models to determine the most suitable one.

2. Exploratory Data Analysis

2.1 EDA for Fps

2.1.1 Distribution for Number of Non-Zero Features per Observation

We begin with a summary of the training data. For fps features, we count the non-zero entries per observation and plot their distribution. As shown in fig. 2.1, each training sample has fewer than 100 non-zero values out of 1024 features, indicating high sparsity in the feature matrix.

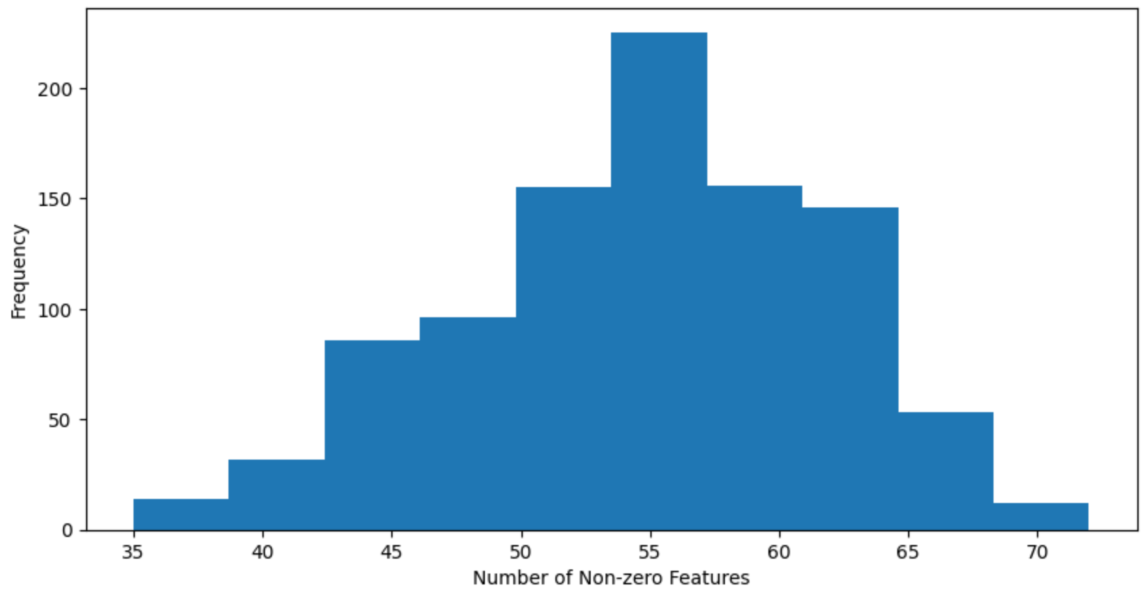


Figure 2.1: Distribution of the Number of Non-Zero Features for All Observations

2.1.2 Summary Statistics for Each Feature

We analyzed the distribution of each feature, calculating the proportion of non-zero observations per feature, shown in fig. 2.2. Most proportions fall within [0, 0.03], indicating an extremely imbalanced number of 0s and 1s. While many features are near zero (blue), a few are around 0.5 (white) or approach one (red/orange), highlighting the need for standardization in scale-sensitive methods like PCA. Additionally, 57 features are constant (all 0s or 1s) and may be removed via dimensionality reduction.

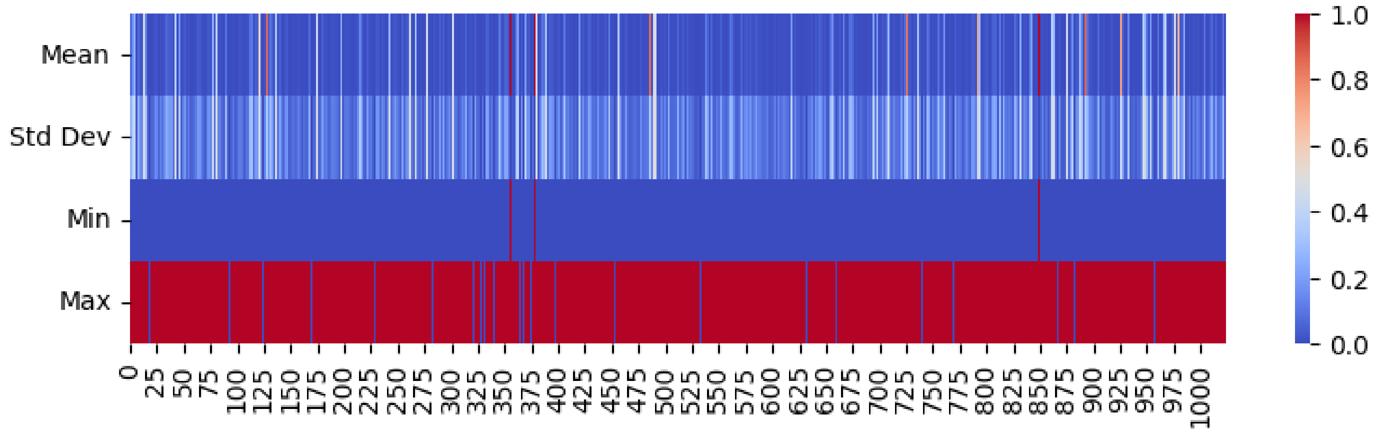


Figure 2.2: Proportions of Non-Zero Observations for Each Fps Feature

2.1.3 Suitablility for Applying PCA

Before applying the dimension-reduction techniques, we also check the correlations between features. From fig. 2.3, we observe several highly correlated features (red or blue dots excluding diagonals). To address this, PCA may be applied to reduce multicollinearity. Besides reducing multicollinearity, PCA helps address high dimensionality in our fps features. Since the number of features (p=1024) exceeds the sample size (n=975), some methods may suffer from interpolation. For instance, in linear regression without penalty terms, if $p > n$, X^TX is not invertible and we cannot use $(X^TX)^{-1}X^Ty$ to derive optimized coefficients. PCA addresses this by retaining 95% of the variance with only 410 principal components (PCs) in the fps dataset.

2.2 EDA for Embed Features

2.2.1 Summary Statistics

From fig. 2.4, we observe that most features exhibit similar ranges and variability, with standard deviations and means clustering at relatively consistent levels. However, a subset of features (orange/red hues) demonstrates notably larger scales, indicating higher variability than the majority. This suggests that normalization is necessary before model training, particularly when applying PCA, as it is sensitive to differences in feature magnitudes.

Additionally, as before, constant features will be removed as part of feature preprocessing to improve model efficiency.

2.2.2 Suitability for PCA

To explore feature relationships, we compute the correlation matrix and visualize it as a heatmap (fig. 2.5). The presence of highly correlated features (deep red/blue areas) indicates potential redundancy, as some features convey overlapping information. Similar to the preprocessing for FPS, we will apply PCA to address the multicollinearity between features and reduce dimensionality. As shown in the variance explained plot, fig. 2.6, 93 out of 224 components can explain over 98% of the total variance in the dataset. Therefore, we can replace the 224 features with 93 PCs to improve model efficiency without substantial information loss.

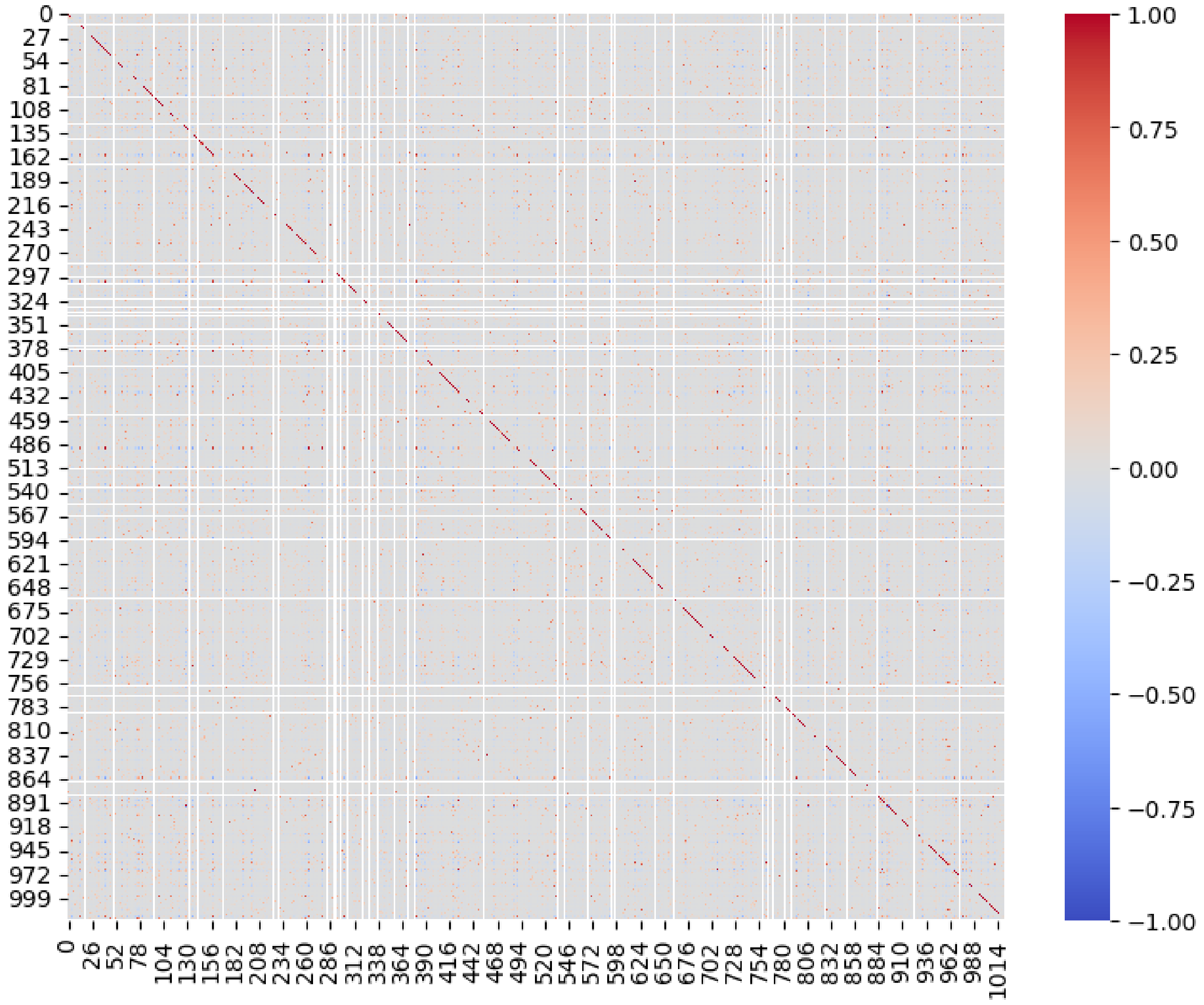


Figure 2.3: Correlation between All Fps Features

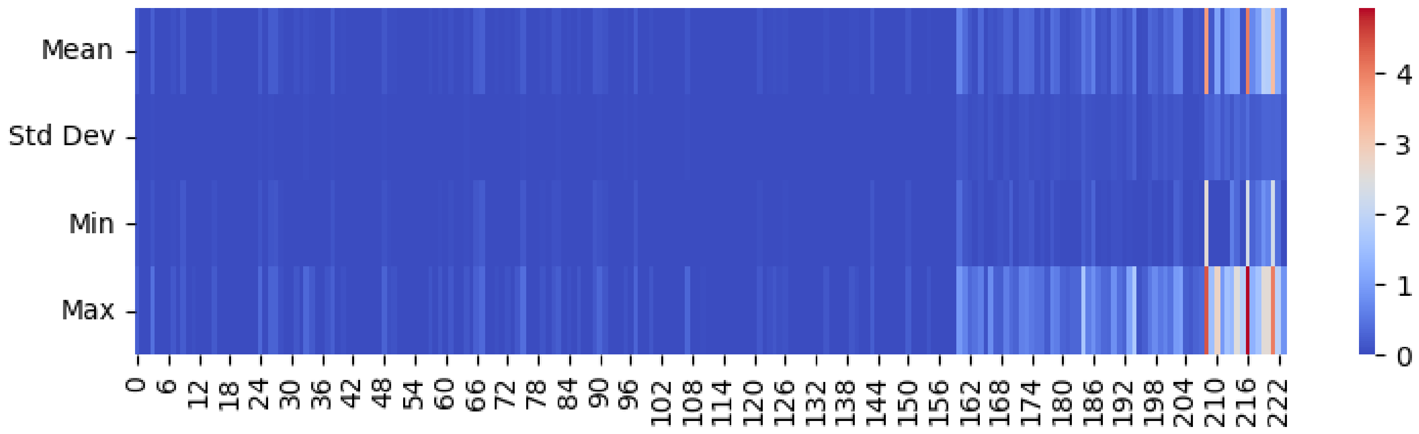


Figure 2.4: Heatmap for Selected Embed Summary Statistics

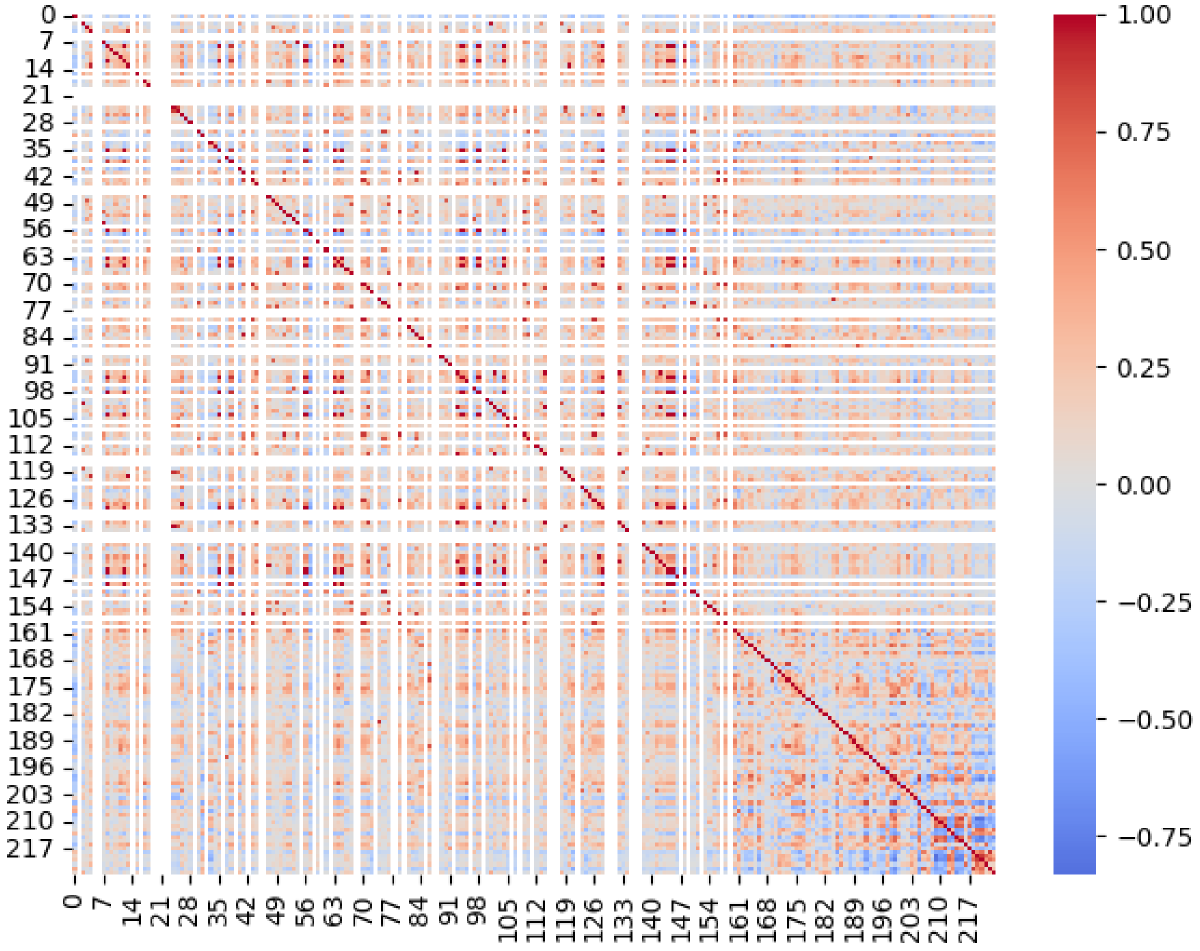


Figure 2.5: Embed Feature Correlation Heatmap

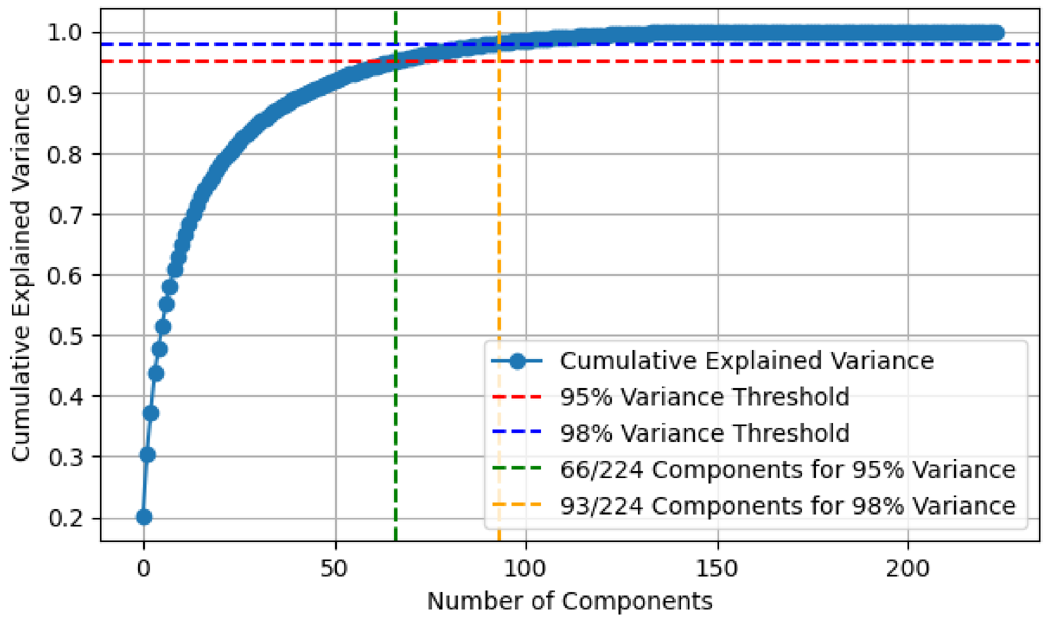


Figure 2.6: Variance Explained by Components in PCA for Embed Data

2.3 EDA for Binding Affinity

Since binding affinity is continuous, we visualize its distribution using a histogram with a kernel density estimate (KDE) (fig. 2.7). The distribution is slightly left-skewed and unimodal, with a peak around 7.5-8, indicating most

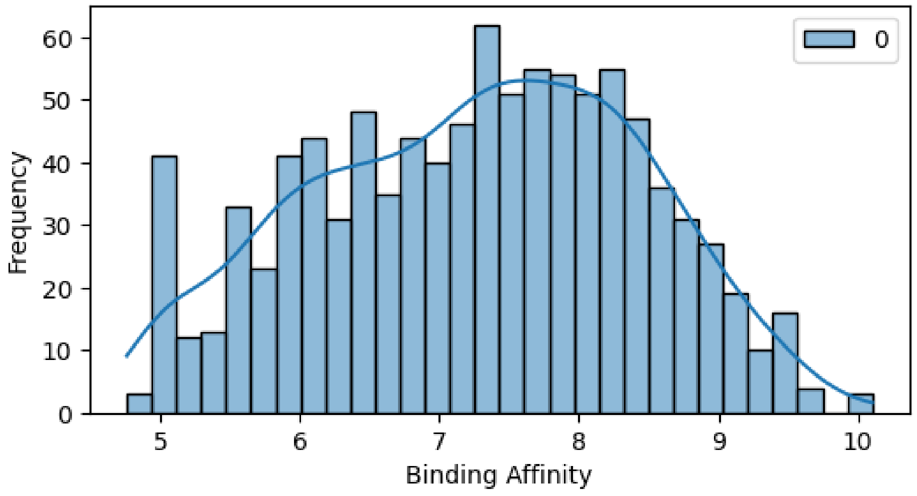


Figure 2.7: Distribution of Binding Affinity

molecules tend to have binding affinity values within this range. Some potential outliers appear at two ends, particularly near 5 and 10, but are not extreme.

As the distribution does not exhibit severe skewness or multiple modes, no transformation (e.g., log or Box-Cox) is needed for this target. Binding affinity values will be used as-is for modelling.

3. Comparison and Construction of Models

3.1 Model Structure, Candidates and Fitting

General structure. Putting together all aforementioned feature engineering and future modelling, the entire pipeline[1] before fitting the model is included in Table 3.1: For training and hyperparameter selection, we use

Table 3.1: Pipeline of Pre-processing (executed line by line)

	Lasso	Ridge	KNN (Fps)	KNN (Embed)	RF	NN
Remove Constant Features	✓	✓	✗	✓	✓	✓
Standardisation	✓	✓	✗	✓	✓	✓
PCA	✓	✓	✗	✓	✓	✓
Box-Cox Transformation	✓	✓	✗	✗	✗	✗

5-fold cross-validation to mitigate overfitting and better estimate model generalization performance by leveraging multiple subsets of the training data. Box-Cox transformations are applied only to models assuming normality.

[1]Variations explained in PCA are varied by models and datasets.

Candidate models. We choose to fit four different regressors using the fps and embed features:

1. **Regularized Linear Regression.** We consider the **Lasso** and **Ridge** regression, with L1 and L2 term respectively. The hyperparameter to tune is the regularization strength α and the proportion of variance retained through PCA. For lasso regression, we tune $\alpha \in (10^{-4}, 10)$ for fps dataset and $\alpha \in (10^{-3}, 10^{-1})$ for embed dataset. For ridge regression, we tune $\alpha \in (10^{-1}, 10^{4})$ for fps data and $\alpha \in (150, 200)$ for embed data.

   The range chosen reflects the large number of features (and hence regression coefficients), and the need for strong regularisation since L2 penalty does not remove features.

   The MSE curves for α tuning are shown in figs. 3.1a to 3.1d. All four plots exhibit a characteristic U-shape, where the CV mean MSE is minimized at an intermediate value of α. When α is too small, regularization is weak, allowing large coefficients and leading to high estimation error. Conversely, when α is too large, both Lasso and Ridge overly shrink the coefficients. The optimal α balances this trade-off between estimation and approximation error, minimizing overall risk. The selected parameters for each model and dataset are summarized in table 3.2.

   Table 3.2: Optimal Hyperparameters for Linear Regressors on Fps and Embed Datasets

Model	Parameters	Fps	Embed
Lasso	α	0.023	0.011
	Variance Retained	95%	98%
Ridge	α	885.867	165.789
	Variance Retained	95%	98%

2. **K-Nearest Neighbors (KNN) Regressor.** KNN assumes that points close in feature space have similar target values. As it makes no normality assumption, we omit Box-Cox transformations.

   For fps features, which are binary, the Hamming distance[2] is a natural choice of dissimilarity. For embed features, which are continuous and often highly correlated, the standard Euclidean distance is used.

   The main hyperparameters are the number of neighbours K, whether to use uniform or distance-based weighting in the prediction, and the proportion of variance retained through PCA (embed features only, due to high correlations). The tuning results are shown in figs. 3.2a and 3.2b. The optimal parameters for each model and dataset are summarised in table 3.3.

   Table 3.3: Optimal Hyperparameters for KNN Model for Fps and Embed Dataset

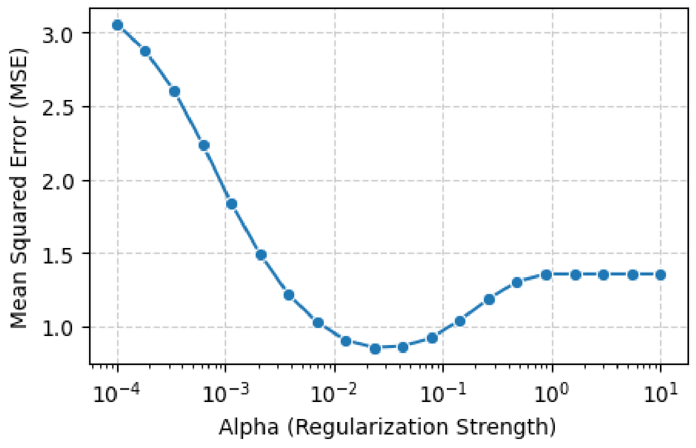
Parameters	Fps	Embed
K	5	7
Weighting Method	Weighted Average	Weighted Average
Variance Retained	(Not Applied)	95%

3. **Random Forest (RF) Regressor.** For the RF model, we tune two main hyperparameters: the number of trees in the ensemble and the proportion of variance retained through PCA. The optimal settings for each dataset are reported in table 3.4.

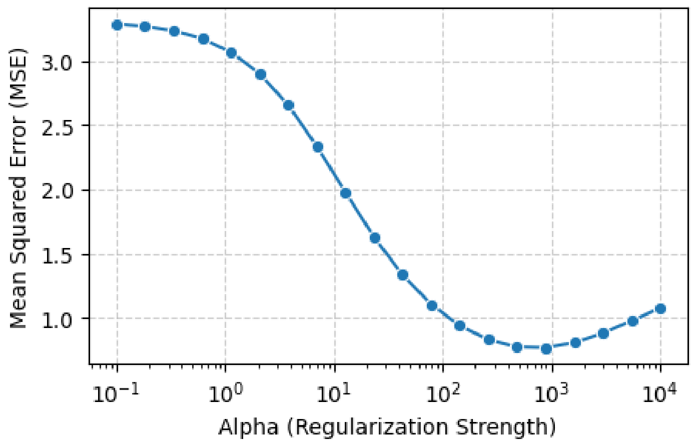
4. **Neural Network (NN).** Simple neural networks, i.e., Multilayer Perceptron (MLP), are used here. The model will be trained by stochastic gradient descent. To prevent overfitting, we will use early stopping.

   Hyperparameters are the number of layers and units per layer, activation (ReLU or logistic), and proportion of variance retained through PCA. The optimal settings for each dataset are reported in table 3.5.

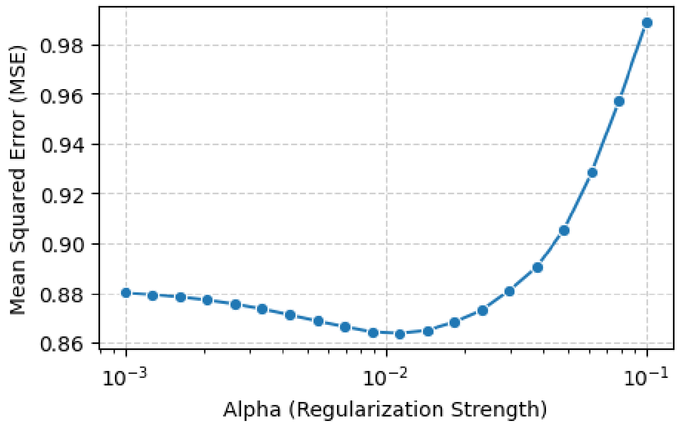
[2]The Hamming distance between two equal-length binary vectors is the proportion of disagreeing components in the two.



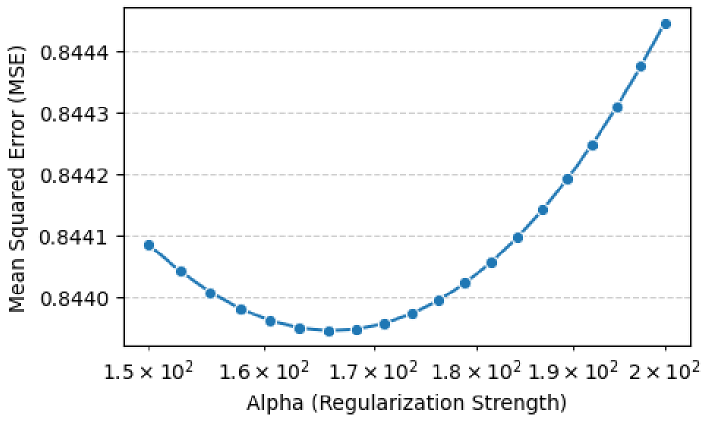
(a) Lasso Regression for Fps Feature: MSE vs. α



(b) Ridge Regression for Fps Feature: MSE vs. α

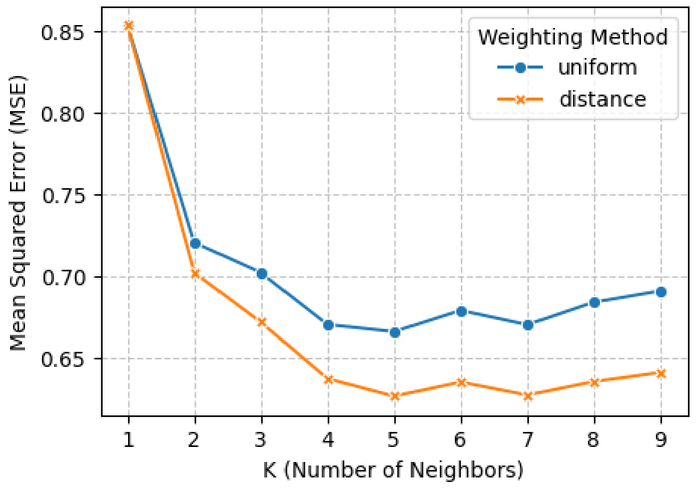


(c) Lasso Regression for Embed Feature: MSE vs. α

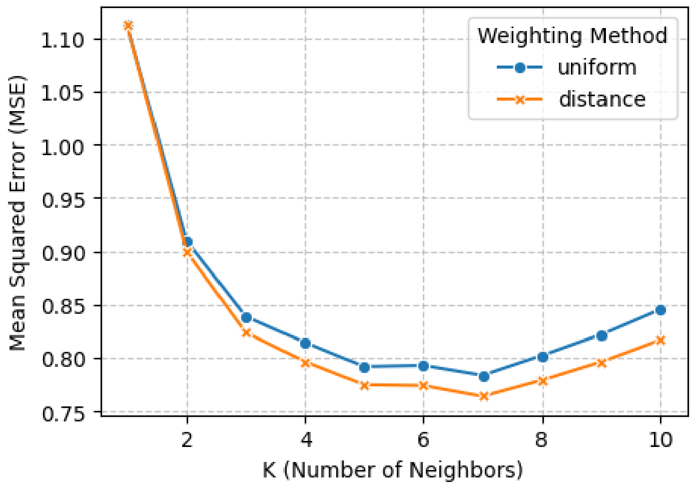


(d) Ridge Regression for Embed Feature: MSE vs. α

Figure 3.1: Linear Regression: MSE vs. α



(a) KNN for Fps Features



(b) KNN for Embed Features

Figure 3.2: KNN: MSE vs Number of Neighbors K

Table 3.4: Optimal Hyperparameters for RF Models on FPS and Embed Datasets

Parameters	Fps	Embed
Number of Trees	200	400
Variance Retained	85%	90%

Table 3.5: Optimal Hyperparameters for NN Models on FPS and Embed Datasets

Parameters	Fps	Embed
Number of Layers	2	1
Number of Layers	128, 128	64
Activation Function	Logistic	Logistic
Variance Retained	95%	85%

3.2 Model Selection

The model is selected based on the CV MSE, model assumptions, interpretability and computation cost (i.e., fitting time, model evaluation time, and memory usage).

3.2.1 Model Performance

Figure 3.3 displays the mean CV MSE for each fps model, along with a 95% confidence interval. We can see that

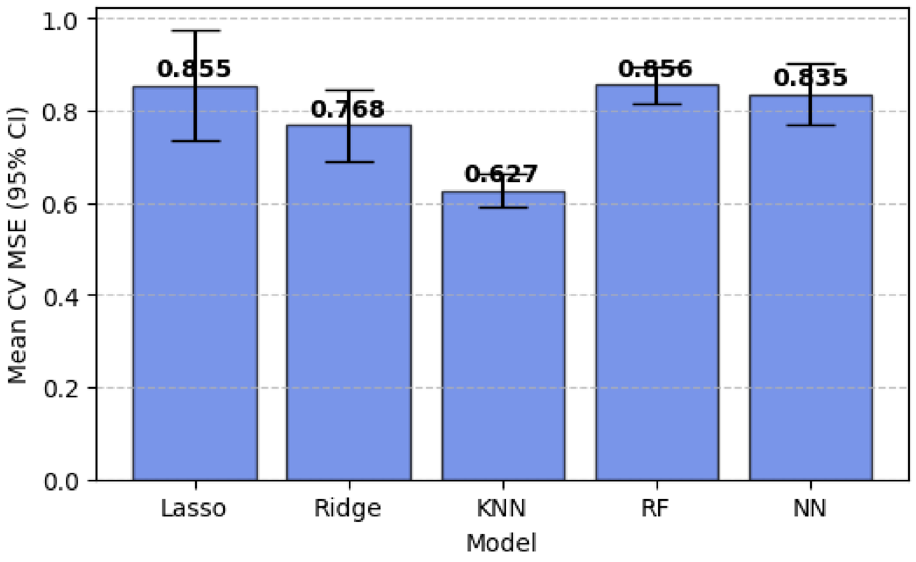


Figure 3.3: Performance Comparison for fps Models

KNN performs the best with the lowest MSE at 0.627, followed by Ridge Regression (MSE = 0.768). Lasso, RF and Neural Network do not perform as well on the training set, all with high MSE of more than 0.83. From this, KNN is the most promising fps model.

For the embed model, according to fig. 3.4, Lasso, Ridge, Random Forest, and Neural Network models have similar performance, with MSE values more than 0.83. In contrast, KNN achieves the lowest MSE (0.764), outperforming all other models. Based on these results, KNN is the most promising embed model.

3.2.2 Model Assumptions and Computational Costs

Model assumptions and interpretability. The linear regression models (Lasso and Ridge) rely on standard assumptions: linearity, independence of predictors, homoscedasticity, and normally distributed errors. In contrast, the other models, KNN and RF, make fewer statistical assumptions, which is generally better. KNN assumes that nearby observations in feature space have similar target values, while Random Forests make predictions based on recursive partitioning via decision trees.

For interpretability, Lasso and Ridge are the most straightforward, providing direct insight into the relationship between features and the response through their coefficients. RF models offer limited interpretability through feature importance scores. KNN, as a non-parametric model, offers minimal interpretability beyond local similarity.

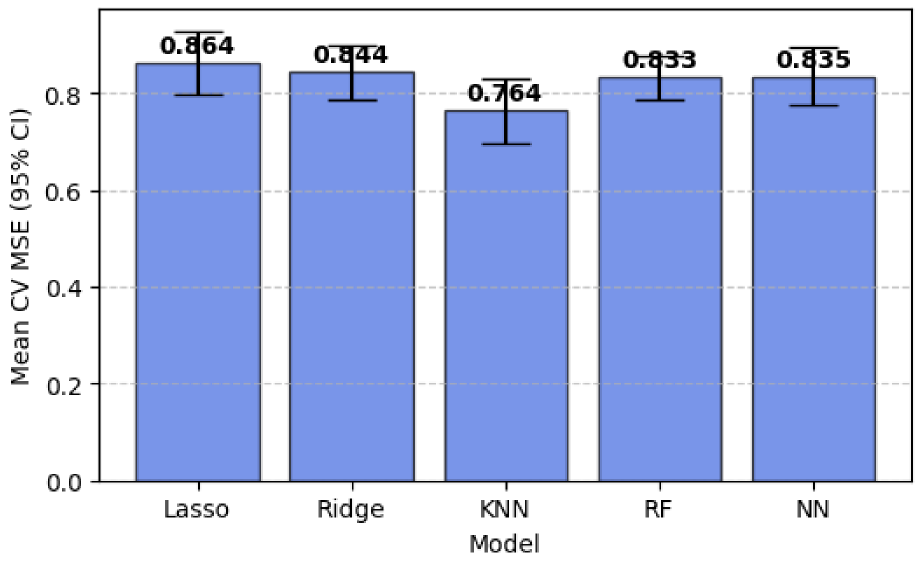
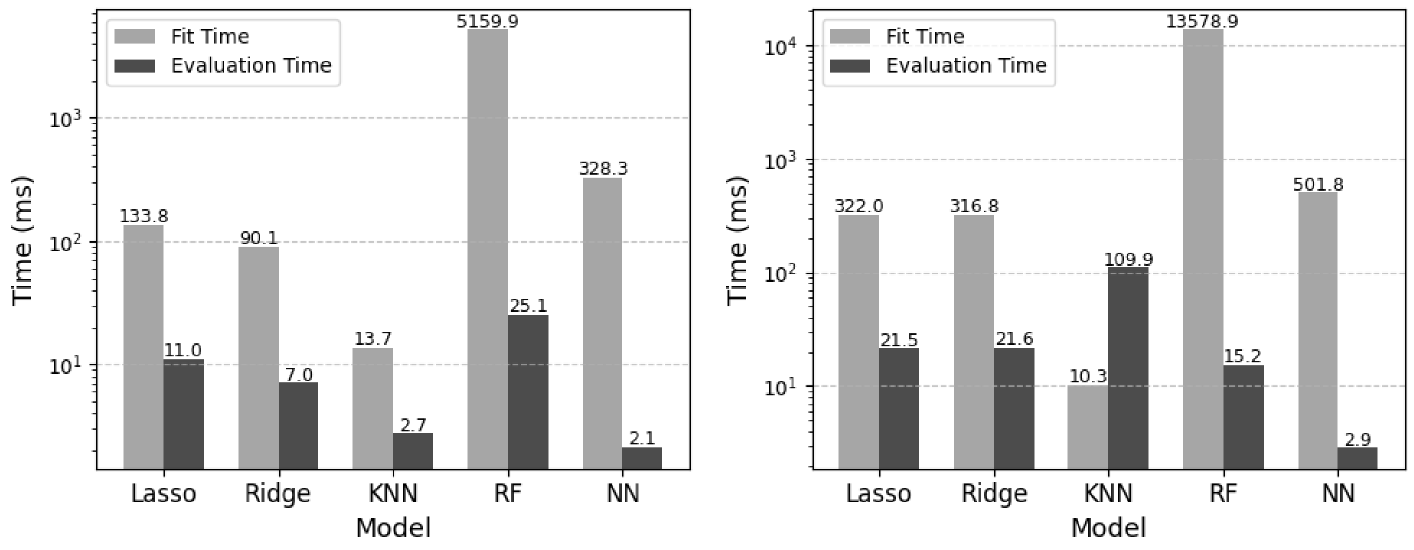


Figure 3.4: Performance Comparison for Embed Models

Computational cost. We now examine the computational cost to finalize model selection. Training and evaluation time for different models on two datasets are presented in fig. 3.5. Across both datasets, the computational



(a) Models on Embed Features

(b) Models on Fps Features

Figure 3.5: Training Time and Evaluation Time (ms) For Different Models

cost trends remain consistent. Random Forest is the most time-consuming model to train, requiring around 5000 ms and more than 10000 ms for the embed and fps datasets, respectively. KNN, in contrast, has the shortest training time, taking only around 14 ms and 10 ms for embed and fps datasets, making it highly efficient.

For evaluation time, the trend differs. On the embed dataset, KNN remains the fastest, taking only around 3 ms. However, on the fps dataset, KNN becomes the slowest (more than 100 ms), though not significantly. This is primarily due to the computation of the Hamming metric, which is computationally more expensive than Euclidean distance.

The memory requirements for different models are summarized in table 3.6. Given the dataset size, KNN may become memory intensive for large-scale applications. However, in our setting, the trade-off in memory is reasonable given its superior predictive performance. Considering all aforementioned model performance, assumptions, interpretability, and computation cost (as heuristically outlined in table 3.7), we select the KNN as the final model

Table 3.6: Theoretical Memory Complexity for Each Model

Model	Space Complexity	Memory Allocation
Lasso/Ridge	$\mathcal{O}(p)$	Stores a vector of coefficients
KNN	$\mathcal{O}(np)$	Stores the full dataset for inference
Random Forest	$\mathcal{O}(Tpd)$	Stores multiple decision trees
NN (MLP)	$\mathcal{O}(Lp)$	Stores weights of hidden layers

* n: dataset size, p: feature size, T: number of trees, d: tree depth, L: number of layers.

Table 3.7: Performance, Assumptions and Computational Cost of Each Model

Model	Lasso	Ridge	KNN	RF	NN
Performance (MSE)	Good	Good	Best	Good	Good
Assumptions	Many	Many	Minimal	Minimal	Minimal
Interpretable	Yes	Yes	No	Limited	No
Computational Cost (Fitting Time)	Good	Good	Best	Worst	Moderate
Computational Cost (Evaluation Time)	Good	Good	Moderate	Good	Good
Computational Cost (Memory Usage)	Good	Good	Moderate	Worst	Worst

for predicting binding affinity. The final model is formulated as follows.

$$\hat{y}(\widetilde{x}) = \frac{\sum_{i\in \mathrm{knn}(\widetilde{x})} w_i y_i}{\sum_{i\in \mathrm{knn}(\widetilde{x})} w_i}, \quad w_i = \frac{1}{D(\widetilde{x}, \widetilde{x}_i) + \epsilon}$$

where i is the index of k nearest neighbours of $\widetilde{x}$ (5 for fps data, 7 for embed data), ϵ is a small constant to prevent division by zero, and the dissimilarity metric is given by

$$D(u, v) = \begin{cases} \frac{\sum_{j=1}^{p_{\text{fps}}} \mathbb{I}_{u_j \neq v_j}}{p_{\text{fps}}}, & \text{for fps features, i.e., } u, v \in \{0,1\}^{p_{\text{fps}}}; \\ \|u - v\|_2^2, & \text{for embed features, i.e., } u, v \in \mathbb{R}^{p_{\text{embed}}}. \end{cases}$$

Note that $\widetilde{x}$ is the data transformed after the aforementioned preprocessing as in table 3.1.

3.3 Analysis of Performance

3.3.1 Fps Dataset

Table 3.8a shows the performances of the final KNN model for our FPS dataset. As we do not use the public test set when fitting our models, it is safe to use the test MSE as an estimator of the MSE on the new private dataset. CV MSE can also be used due to the nature of CV, where we generated independent test sets in each iteration (though the fold number may influence its accuracy). We can see from table 3.8a that the test MSE (0.825) and MAE are higher compared to the CV MSE (0.627) and MAE, suggesting potential overfitting. Meanwhile, we can see from the True vs Predicted plot in fig. 3.6 that our predictions are generally close to the true values, though some of the variations in true binding affinity are left unexplained as indicated by the linear trend in the residual plot (right). The model works well for true binding affinity with a medium size, while systematically underestimating big binding affinity values and overestimating the small binding affinity values. This may be because we are not including all features that affect binding affinity in our model.

3.3.2 Embed Dataset

According to table 3.8b, the CV and test MSE values (0.764 vs. 0.746) and MAE values (0.683 vs. 0.706) are consistent, suggesting that overfitting is not a potential concern. Therefore, this embed KNN model is robust for making predictions. Apart from MSE and MAE values, the True vs. Predicted plot (see fig. 3.7) indicates that

(a) Performance of KNN on Fps Datasets

Metrics	Average across CV Test	Public Test
MSE	0.627	0.825
MAE	0.594	0.744

(b) Performance of KNN on Embed Datasets

Metrics	Average across CV Test	Public Test
MSE	0.764	0.746
MAE	0.683	0.706

Table 3.8: KNN Model Performance on Fps and Embed Datasets

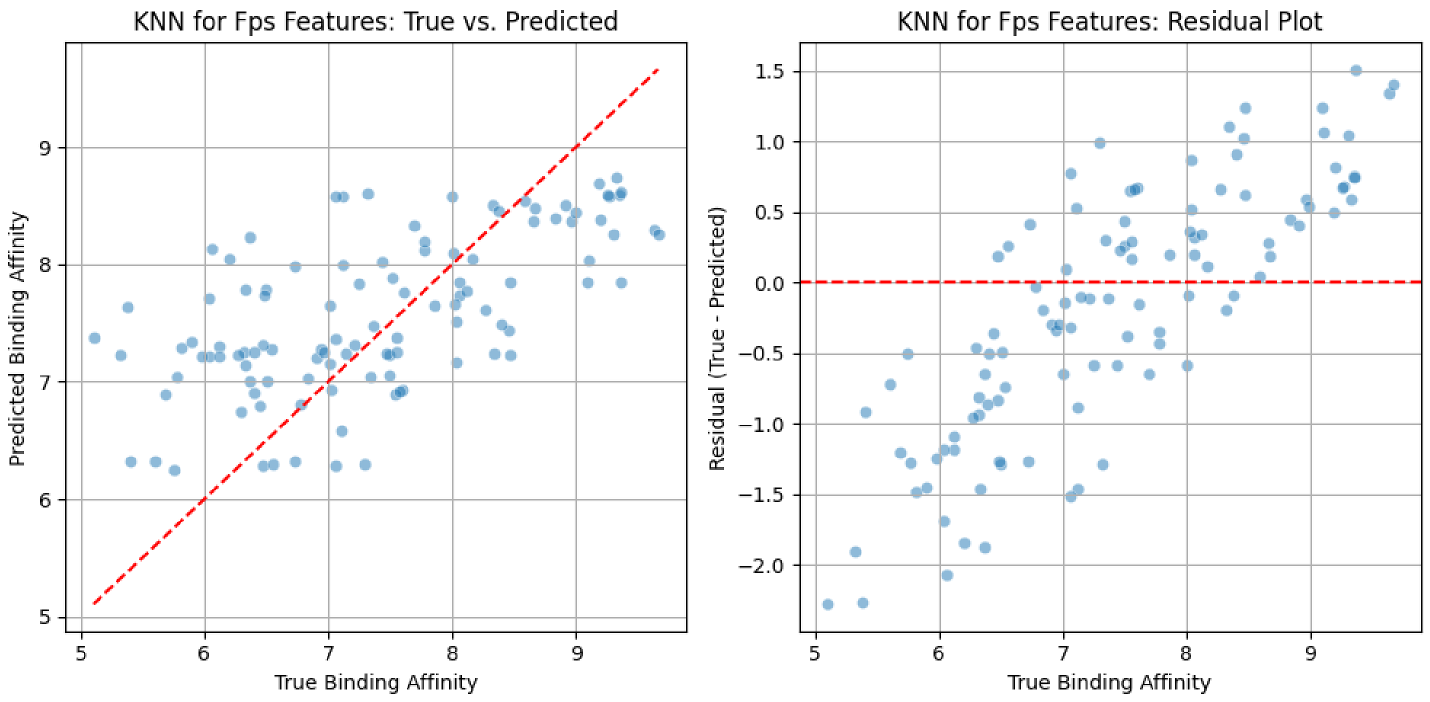


Figure 3.6: KNN for Fps Features: Analysis of Performance

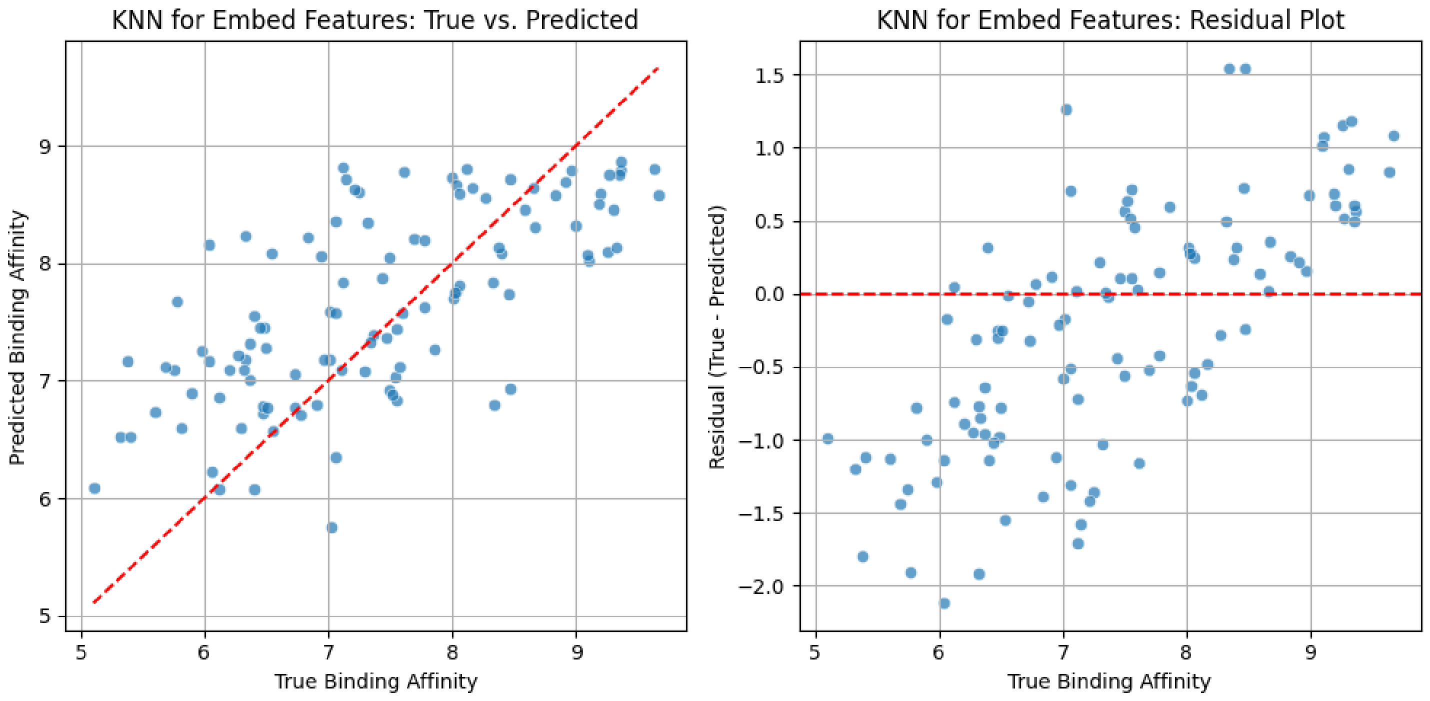


Figure 3.7: KNN for Embed Features: Analysis of Performance

the model generally aligns well with the data, with all points lying close to the diagonal. However, the residual plot reveals a linear trend similar to the fps residual plot, indicating potential bias. The reason may be that some variances are left unexplained when only including embed features, and we may consider utilizing both fps and embed features to make predictions.

4. Final Model Performance and Predictions

4.1 Combining the Features

For the two distinct feature sets, we adopt a concatenation-based approach, enabling the model to leverage information from both representations simultaneously. The preprocessing steps remain consistent with the previous sections:

$$\left.\begin{array}{l} \begin{bmatrix} \text{fps}_1 \\ \vdots \\ \text{fps}_{1024}\end{bmatrix} \longrightarrow \text{Standardisation} \to \text{PCA} \xrightarrow[\text{Only for Linear Model}]{\text{Box-Cox Transformation}} \\ \begin{bmatrix} \text{embed}_1 \\ \vdots \\ \text{embed}_{224}\end{bmatrix} \longrightarrow \text{Standardisation} \to \text{PCA} \xrightarrow[\text{Only for Linear Model}]{\text{Box-Cox Transformation}} \end{array}\right\} \xrightarrow{\text{Concatenate}} \text{Combined Data } \widetilde{X}.$$

4.2 Choice of Final Model

Since KNN performed best on both embed and fps features individually, we first applied it to the combined dataset. The best KNN model (via CV) achieves a CV MSE of approximately 0.7 – better than on embed alone (≈ 0.764) but worse than on fps (≈ 0.627, but with overfitting). Moreover, several concerns arise when using KNN on the combined dataset:

1. After concatenating the two datasets, the feature dimension becomes significantly larger than the dataset size ($224 + 1024 \gg 975$). Even after PCA, some features may contribute less to predictive performance, yet they cannot be easily removed alone.

2. The choice of distance metric in KNN becomes questionable. While Euclidean distance may work well within a single feature space, it may be inappropriate for a dataset combining two feature types (continuous embed features and binary fps features). The dissimilarity measure might no longer reflect the true relationships in the data.

These concerns motivate the need for an alternative approach, one that incorporates stronger regularization and does not rely on potentially unreliable distance metrics. Elastic Net provides a suitable solution, as it balances L1 (sparsity) and L2 (ridge) penalties, allowing the model to handle high-dimensional data effectively. The elastic net is formulated as

$$\hat{\beta} = \arg\min_{\beta} \left\{ \sum_{i=1}^{n} \left\| y - \widetilde{X}\beta \right\|_2^2 + \alpha \left[(1-\lambda) \|\beta\|_2^2 + \lambda \|\beta\|_1 \right] \right\}$$

where $\left(\widetilde{X}, y\right)$ is the concatenated dataset, α and λ balance the strength of L1 and L2 penalty. In practice, the Elastic Net model achieves a CV MSE of 0.675, already outperforming the KNN model (0.692). Furthermore, as shown in table 4.1, the MSE on the public test set reaches 0.683, marking a notable improvement over the previous KNN models fitted on each dataset individually. This improvement suggests that the issue of overfitting has been mitigated. This makes Elastic Net our final choice for predicting binding affinity. Additionally, the effect

Table 4.1: Performance of Elastic Net on Combined Dataset

Mean MSE across CV Test	MSE on Public Test
0.675	0.683

of regularisation is shown in fig. 4.1, compared to OLS, we noticed that the coefficients are largely shrunk or even removed from the model.

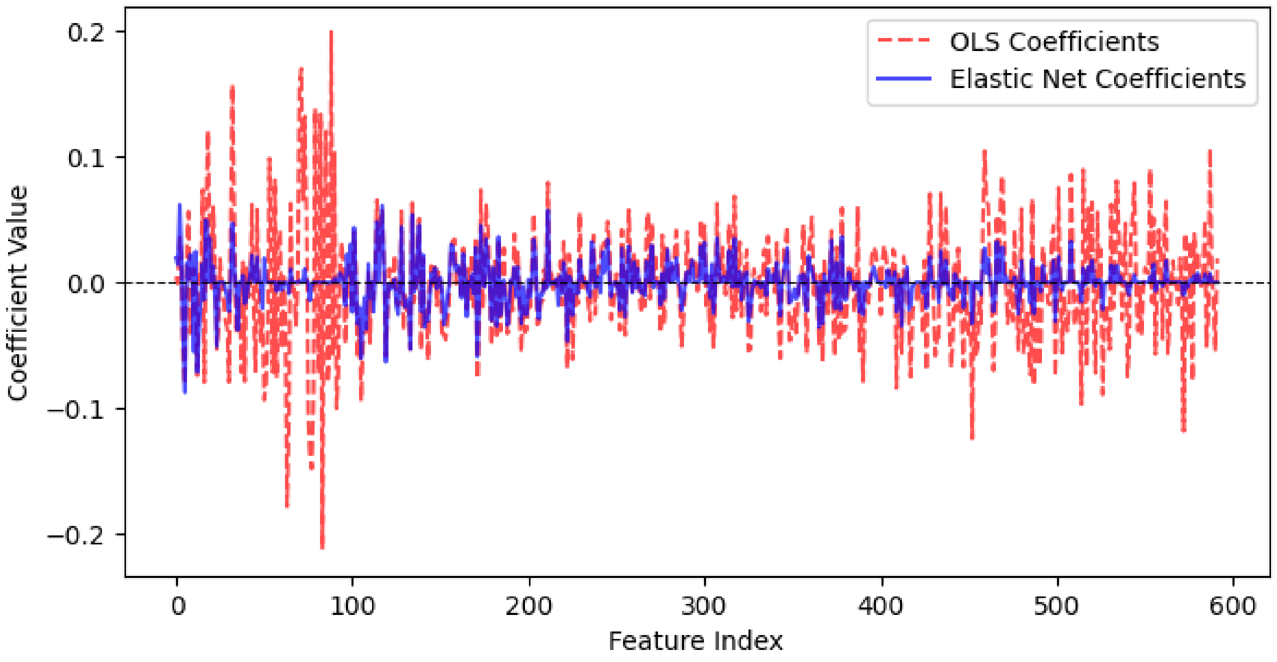


Figure 4.1: Effect of Elastic Net Regularisation Compared to OLS

4.3 Expected Performance on Private Set

Finally, we analyze the expected performance on the private set. Since we don't have true labels, a reasonable estimate of expected performance is using the CV error distribution from training.

$$\widehat{\text{MSE}}_{\text{private}} = \frac{1}{K}\sum_{k=1}^{K} \text{MSE}_k = 0.675$$

where $K = 5$ is the number of folds in CV, and the uncertainty can be estimated by a 95% confidence interval from CV

$$\text{CI} = \widehat{\text{MSE}}_{\text{private}} \pm z_{0.025}\frac{\sigma}{\sqrt{K}} = [0.616, 0.735]$$

where σ is the standard deviation of CV MSE. Meanwhile, as the fps and embed distributions are similar on private and public test sets, we can use public test MSE to estimate performance on the private set. The bootstrap CI (resampled 1000 times) for test MSE is $(0.499, 0.859)$, with a similar mean and a slightly bigger range to the CV MSE. These results suggest that the model exhibits consistently good performance, with stable CV and test MSEs aligning with each other.

5. Conclusion

In this project, we explored multiple regression models to predict molecular binding affinity based on two sets of features, fps and embed. After preprocessing, model selection, and performance evaluation, a combined Elastic Net model provides the best balance between predictive accuracy and generalisability, which was used to obtain predictions on the private test set in a separate CSV file. It has good CV performance, interpretability, computational efficiency and no overfitting issues compared to other models. However, residual plots still show a linear trend, suggesting minor assumption violations. Future work could incorporate additional factors influencing binding affinity, or a combination of two dissimilarity metrics in KNN models.

A. Python Code Appendix

```python
# Import libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import norm
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV, KFold
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler
from feature_engine.transformation import YeoJohnsonTransformer
from sklearn.feature_selection import VarianceThreshold
from sklearn.decomposition import PCA
from sklearn.linear_model import LinearRegression, Ridge, Lasso, ElasticNet
from sklearn.ensemble import RandomForestRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.neural_network import MLPRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error
from scipy.stats import bootstrap

# Load the training data and the public test set
X_fps_train, X_embed_train = pd.read_csv('X_fps_train.csv', index_col = 0),
↪ pd.read_csv('X_embed_train.csv', index_col = 0) # training data
y_train = pd.read_csv('y_train.csv', index_col = 0).to_numpy() # training label

X_fps_public_test, X_embed_public_test = pd.read_csv('X_fps_public_test.csv', index_col =
↪ 0), pd.read_csv('X_embed_public_test.csv', index_col = 0) # public test set
y_public_test = pd.read_csv('y_public_test.csv', index_col = 0) # public test label

X_fps_private_test, X_embed_private_test = pd.read_csv('X_fps_private_test.csv', index_col =
↪ 0), pd.read_csv('X_embed_private_test.csv', index_col = 0) # private test set

# Define a 5-fold cross-validation
KFCV=KFold(n_splits=5, shuffle=True, random_state=214)

# Append FPS features to Embed features (column-wise)
X_combined_train = pd.concat([X_embed_train, X_fps_train], axis=1).to_numpy()
X_combined_public_test = pd.concat([X_embed_public_test, X_fps_public_test],
↪ axis=1).to_numpy()
```

```python
X_combined_private_test = pd.concat([X_embed_private_test, X_fps_private_test],
 ↪  axis=1).to_numpy()

# Define preprocessing for embed features
embed_preprocessing = Pipeline([
    ('feature_selection', VarianceThreshold(threshold=0)),  # Remove constant features
    ('scaler', StandardScaler()),  # Normalise embed features
    ('pca', PCA()),  # PCA to reduce dimensions
    ('box-cox transform', YeoJohnsonTransformer()), # Box-Cox transformation
])

# Define preprocessing for fps features
fps_preprocessing = Pipeline([
    ('scaler', StandardScaler()),  # Normalize embed features
    ('pca', PCA()),  # PCA to Reduce dimensions
    ('box-cox transform', YeoJohnsonTransformer()), # Yeo-Johnson (Extended Box-Cox)
     ↪  transformation
])

# Combine preprocessing for embed and fps
combined_preprocessing = ColumnTransformer([
    ('embed_processing', embed_preprocessing, list(range(X_embed_train.shape[1]))),  #
     ↪  Process only for embed features
    ('fps_processing', fps_preprocessing, list(range(X_embed_train.shape[1],
     ↪  X_combined_train.shape[1])))  # Process only for fps features
])

# Define full pipeline
combined_elasticnet_pipeline = Pipeline([
    ('preprocessing', combined_preprocessing),  # Apply preprocessing
    ('regressor', ElasticNet())  # Elastic Net Regression
])

# Define hyperparameter grid
combined_elasticnet_param_grid = {
    'preprocessing__embed_processing__pca__n_components': [0.95, 0.98],  # PCA for embed
    'preprocessing__fps_processing__pca__n_components': [0.95, 0.98],  # PCA for fps
    'regressor__alpha': np.logspace(-1, 0, 5),  # Regularisation strength
    'regressor__l1_ratio': np.linspace(0.02, 0.1, 5)  # Balance between L1 and L2 penalty
}

# Perform GridSearchCV with cross-validation
combined_elasticnet_grid_search = GridSearchCV(
    combined_elasticnet_pipeline,
    combined_elasticnet_param_grid,
    cv=KFCV,
    scoring=['neg_mean_squared_error', 'neg_mean_absolute_error'],
    refit='neg_mean_squared_error', # Use MSE to refit the model
    n_jobs=-1   # Using all processors to run faster
)
# Fit the model
combined_elasticnet_grid_search.fit(X_combined_train, y_train)

# Retrieve best hyperparameters and model
combined_elasticnet_best_model = combined_elasticnet_grid_search.best_estimator_

# Retrieve cross-validation results
combined_elasticnet_cv_results = pd.DataFrame(combined_elasticnet_grid_search.cv_results_)
# Top 10 cross-validation results, with scores and best chosen parameters
combined_elasticnet_cv_results.sort_values('mean_test_neg_mean_squared_error',
↪  ascending=False)[[
    'param_preprocessing__embed_processing__pca__n_components',
    'param_preprocessing__fps_processing__pca__n_components',
    'param_regressor__alpha',
    'param_regressor__l1_ratio',
    'mean_test_neg_mean_squared_error',
    'mean_test_neg_mean_absolute_error'
    ]].head(10)

# Performance on public test dataset
combined_elastic_y_pred = combined_elasticnet_best_model.predict(X_combined_public_test)
print(f'MSE on public test: {mean_squared_error(y_public_test, combined_elastic_y_pred)}')
print(f'MAE on public test: {mean_absolute_error(y_public_test, combined_elastic_y_pred)}')

# CV MSE results
expected_mse_private = -combined_elasticnet_grid_search.best_score_
# Extract standard deviation of test MSE across folds for the best model
std_mse_private = combined_elasticnet_grid_search.cv_results_[
    'std_test_neg_mean_squared_error'
    ][combined_elasticnet_grid_search.best_index_]

# Compute 95% confidence interval
z = norm.ppf(0.975)
ci_lower = expected_mse_private - z * (std_mse_private / np.sqrt(5))
ci_upper = expected_mse_private + z * (std_mse_private / np.sqrt(5))

# Print expected MSE and confidence interval
print(f"Expected MSE on private set: {expected_mse_private:.3f} [{ci_lower:.3f},
↪  {ci_upper:.3f}]")

# Test MSE bootstrapping
# convert to 1D arrays
y_test_combined = y_public_test.squeeze()
def mse_stat(sample_indices):
    # Compute the MSE for each sample_indices
    return mean_squared_error(y_test_combined[sample_indices],
     ↪  combined_elastic_y_pred[sample_indices])

```

```python
# Apply bootstrap (95% confidence interval)
res = bootstrap((np.arange(len(y_public_test)),), mse_stat, n_resamples=1000,
↪  method='basic', rng=412)

# Print estimated test MSE and CI
print(f"Bootstrap Estimated Test MSE: {res.bootstrap_distribution.mean():.3f}")
print(f"95% Confidence Interval using Bootstrap: ({res.confidence_interval.low:.3f},
↪  {res.confidence_interval.high:.3f})")

# Export the predictions on the test data in csv format
private_pred = combined_elasticnet_best_model.predict(X_combined_private_test)
prediction = pd.DataFrame(private_pred, columns=['Prediction'])
prediction.index.name='Id'
prediction.to_csv('myprediction.csv') # export to csv file
```