Steam Review Project

As someone who is extremely invested in gaming, I decided to choose a dataset on Kaggle revolving around Steam reviews. Steam is a video game storefront, and the most popular marketplace to distribute and purchase games for PC players. The dataset (https://www.kaggle.com/datasets/kieranpoc/steam-reviews) contains 100,000,000+ steam reviews, with columns the user id, their owned games, number of reviews, playtime, language, the number of users who voted their review positively, whether their review was positive/negative, etc. I want to use this data to try and create a decision tree ML model to attempt and predict review sentiment—using data like playtime, the number of games owned, the number of reviews the user has left, and whether they purchased the game or received it for free. The review sentiment is the "**voted_up**" column, with a 1 being a positive review, and 0 being a negative one. My goal is ultimately to use this project as an opportunity to see what influences fellow user reviews on Steam, as gaming is a passion of mine. It will be interesting to then afterwards analyze my own profile to identify if my reviews/sentiment for my own collection follows the trends I uncover in the project.

Data Collection

For this step of the project, I worked to download my data, steam reviews, from Kaggle into my VM. Listed are the steps I took:

1. **Initial API Setup and KaggleDataset Download:**

I created an API token for Kaggle and uploaded it to my VM instance. After setting up the Kaggle directory within my Python virtual environment, I downloaded the Steam reviews dataset using the following command:

```
kaggle datasets download -d kieranpoc/steam-reviews
```

2. **Unzipping the Dataset:**

After installing the necessary zip utilities, I unzipped the downloaded file:

```
unzip steam-reviews.zip
```

3. **Google Cloud Storage Bucket Creation:**

I created the my bucket, gamasteamreviews, for storing my project data with the following command:

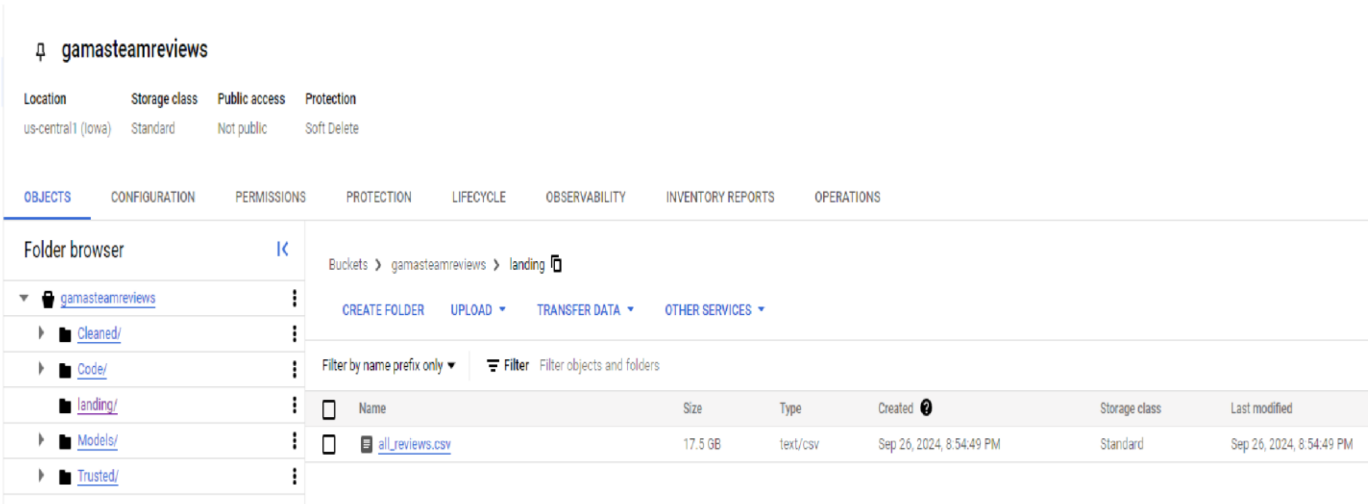
```
gcloud storage buckets create gs://gamasteamreviews --project=gamasteam \ --default-storage-class=STANDARD --location=us-central1 --uniform-bucket-level-access
```

4. **File Upload to Landing Folder:**

I copied the unzipped all_reviews.csv file into the landing folder using this command: `gcloud storage cp all_reviews/all_reviews.csv gs://gamasteamreviews/landing/` Initially, I mistakenly ran the command without a trailing /, which caused the file to be copied as landing instead of placing it into a folder. This led to some troubleshooting before I corrected the issue.

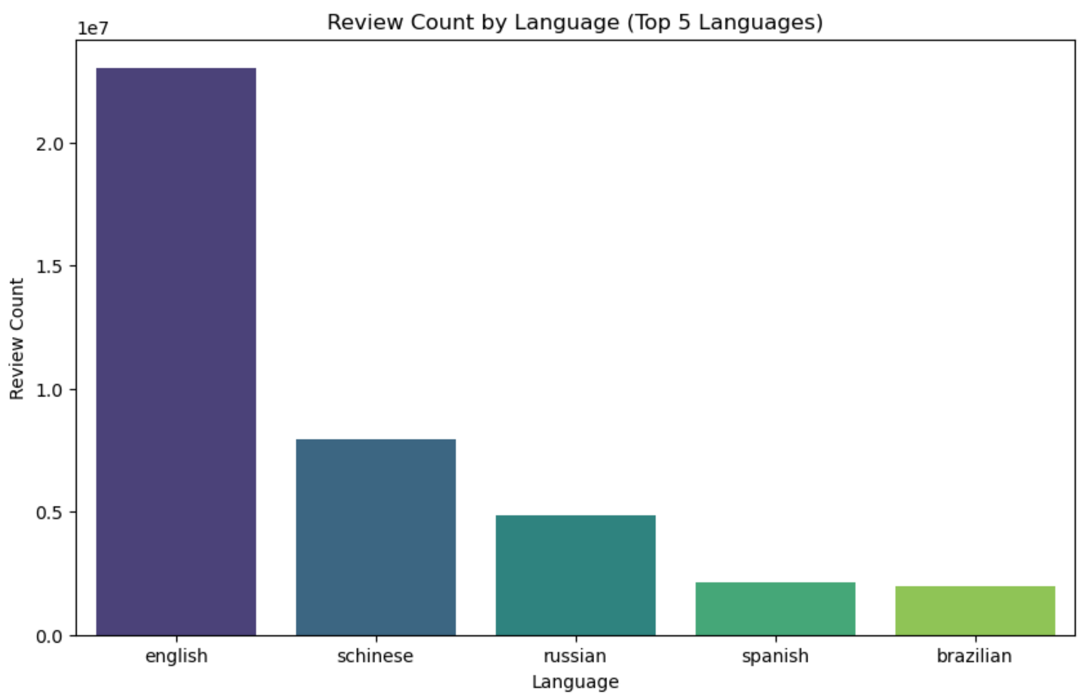
5. **Final Bucket Setup:**

For the last stretch of the milestone, I manually created the additional required folders (cleaned, code, models, and trusted). The final structure of my bucket is shown in the attached screenshot.

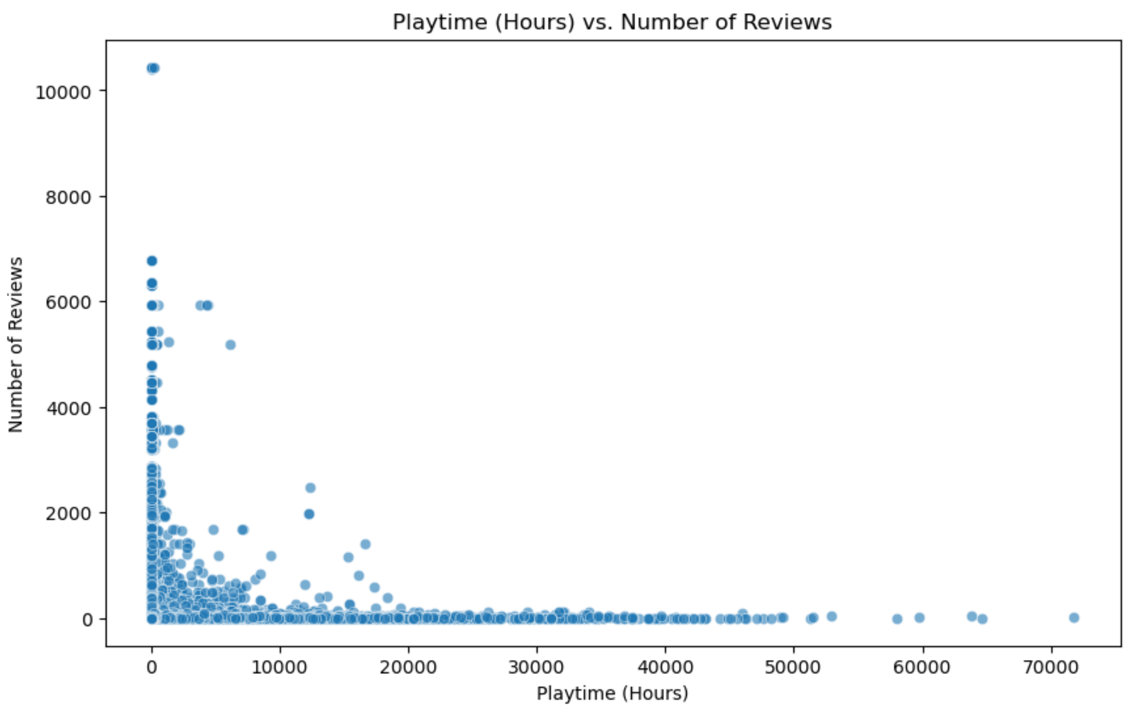


Exploratory Data Analysis and Data Cleaning

In the data analysis portion of this project, I used PySpark to gain a foundational exploratory look at the dataset. Holistically, the dataset comprises a substantial 49,526,668 records. A quick statistical overview revealed some intriguing metrics; for instance, the maximum number of Steam games owned at the time was a whopping 33,345 games (with the minimum, as expected, being 0). Another standout statistic was the highest recorded playtime by a user: 97,317 hours—equivalent to about 4,055 days of playtime. During the exploratory data analysis (EDA), I created two simple graphs to visualize how certain aspects of the data are distributed, shown below:



The overwhelming majority of Steam reviews are written in English, with Simplified Chinese as the next most common language, though the volume of Chinese reviews appears to be roughly one-third of the English reviews. This is pretty interesting as, to my knowledge, China has its own alternatives to Steam that are more accessible due to how strictly Steam is regulated in the Country.



Interestingly, a significant portion of reviews are left by users with playtimes between 0 and 10,000 hours (a broad range). However, it's noteworthy that many reviews come from users with very little playtime—some even showing 0 hours.

As I proceed with feature engineering, one thing I think I'll be wary about is the presence of reviews from users with minimal or no playtime. These low-playtime reviews can mean a variety of things, but they likely don't represent fully developed assessments of the games themselves. More often, these reviews may reflect issues like performance issues or crashes rather than meaningful sentiment about gameplay.

Feature Engineering and Modeling

Now that it's time to begin engineering new features, I created three new ones to capture patterns in reviewer behavior:

- **time_of_day**: This feature uses the review creation timestamp to group reviews into morning, afternoon, evening, and night categories. The idea is that the time of day might influence review sentiment, with gamers potentially leaving more positive reviews during certain times, like in the afternoon or evening.
- **recency_bias**: This feature calculates the number of days between when the user last played the game and when they wrote the review. The assumption is that a shorter gap reflects stronger feelings about the game due to recency bias.
- **played_after_review**: This feature checks if the user continued to play the game after writing their review. If a player keeps playing, it probably indicates a more positive experience.

Below are each of the features used in the model and the treatment they received. Many numerical columns were left as-is, as to my knowledge, decision trees do not need data scaling, typically:

author_num_games_owned	Used as-is
author_num_reviews	Used as-is
author_playtime_forever	Used as-is

author_playtime_last_two_weeks	Used as-is
author_playtime_at_review	Used as-is
author_last_played	Used as-is
recency_bias	Used as-is
played_after_review	Used as-is
language	Used StringIndexer
time_of_day	Used StringIndexer
Timestamp_created	Used as-is

After creating the new features, I filled in any missing values for these new columns using the median of each column to ensure they wouldn't cause issues during model training. Categorical columns like **language** and **time_of_day** were encoded into numerical values using StringIndexer. Finally, I combined all numerical and indexed features into a single features vector using VectorAssembler.

One challenge I faced was deciding which features would be most useful for the model. It was difficult to determine what factors would be most relevant until I shifted my focus to reviewer behavior and emotional state. Drawing from personal experience, I realized that I tend to be kinder to games or movies immediately after finishing them, likely due to recency bias. This insight inspired features like **recency_bias**, which measure the time between when a reviewer last played a game and when they wrote their review. Another challenge was that some new features introduced missing values even though I had already addressed missing data earlier. Diagnosing this issue took significant time because I hadn't initially considered that newly engineered features could create new gaps. This led me to spend a lot of time reviewing my earlier data preparation steps, mistakenly thinking the problem lay in my EDA code. Eventually, I realized the issue stemmed from the feature engineering process itself, and I addressed it by filling in the missing values after creating the new features.

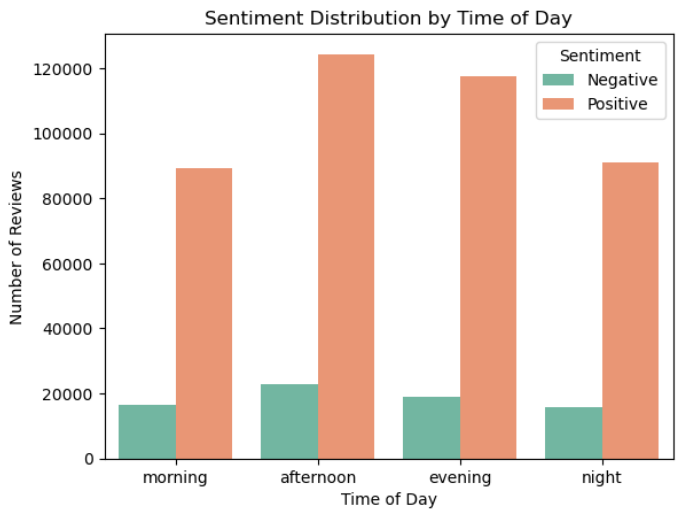
Summary of Outputs:

- **Cross-Validated Accuracy**: 85.61%
- **Precision**: 82.45%
- **Recall**: 85.61%
- **F1-Score**: 80.91%

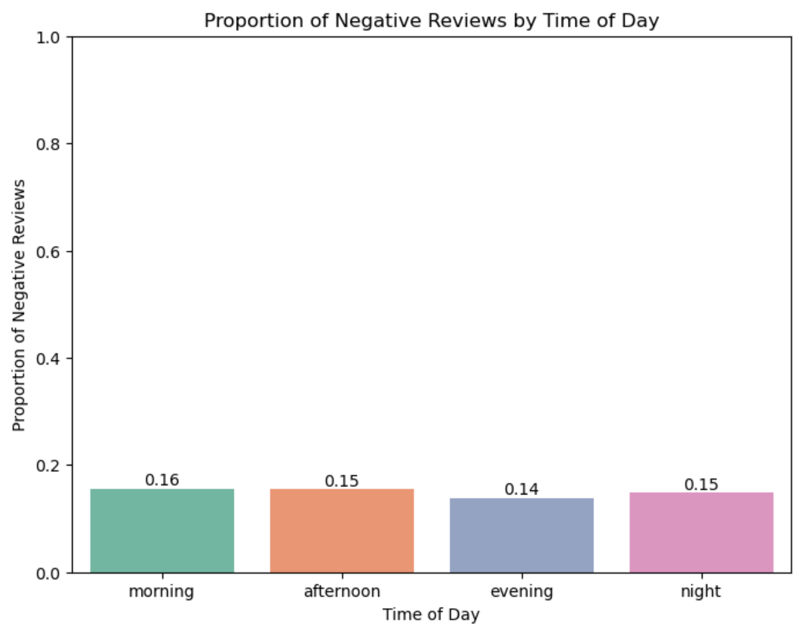
The model did decently well overall. The cross-validated accuracy of 85.61% shows that it correctly classified most reviews as positive or negative during testing. Since the recall is also 85.61%, it means the model was really good at identifying actual positive reviews. The precision was a bit lower at 82.45%, which means that some reviews predicted as positive were actually negative. This tells me the model might sometimes mistake negative reviews for positive ones, which could be an issue if precision is really important. Finally, the F1-score of 80.91% shows that the model has a good balance between precision and recall. This makes it pretty reliable overall for predicting review sentiment. While there's definitely room to improve, these results show that the model is working competently.

Data Visualization

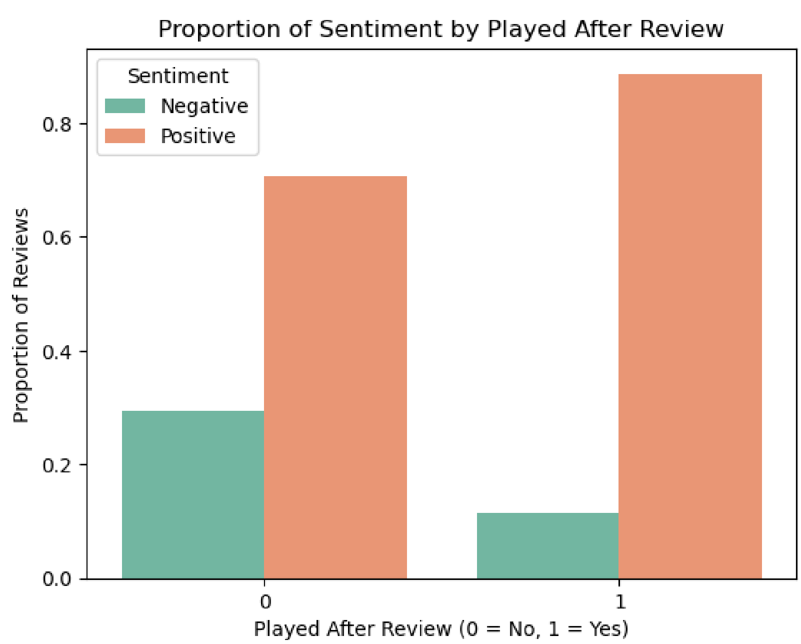
To better understand the data and uncover the biggest influences on review sentiment for my decision tree model, I created a few visualizations.



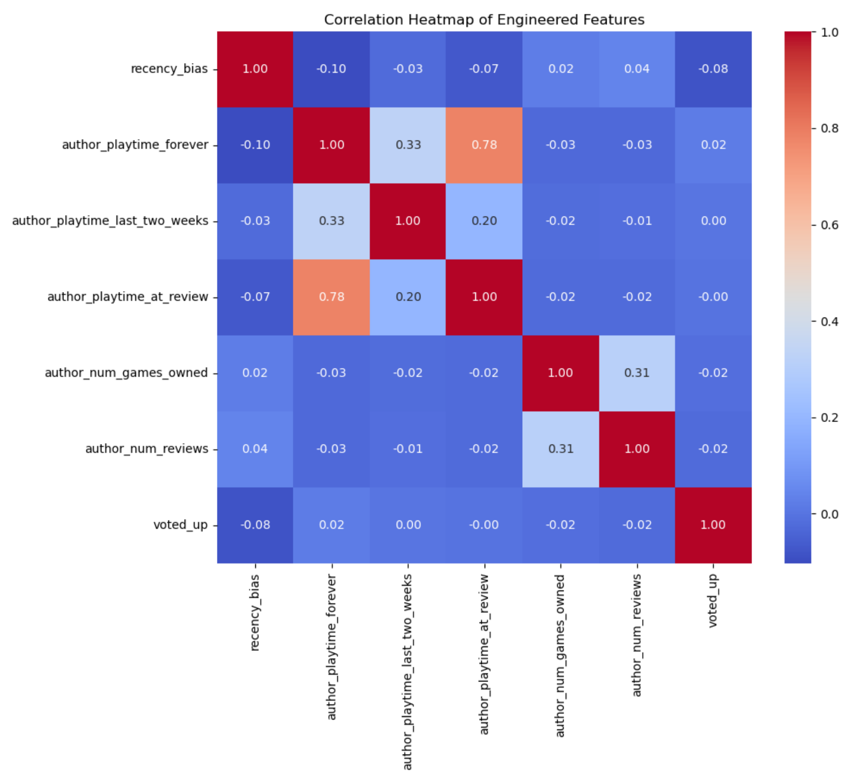
First, I looked at how the time of day affects sentiment. Positive reviews dominate across all time periods, with afternoons and evenings having the most reviews overall. Mornings and nights have fewer reviews, which isn't shocking for mornings, but nights being that low was a bit unexpected. This tells us that time of day might not just impact how many reviews are left but also their tone, with afternoons and evenings being prime times for positivity.



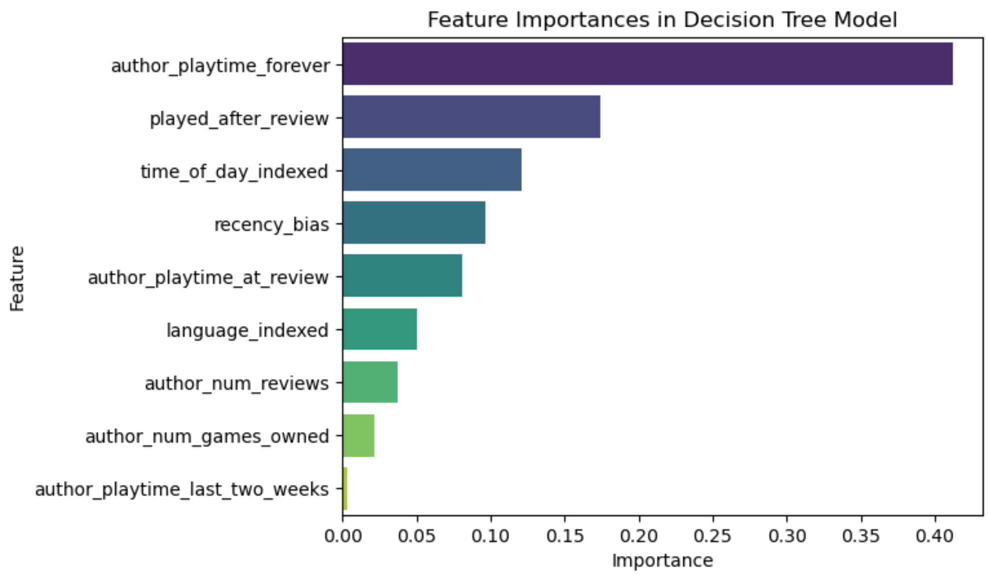
Building on that, I checked the proportion of negative reviews for each time of day. Evenings had the lowest negativity, while mornings had the highest, although the difference was small (14% to 16%). Still, given the massive volume of reviews, it's clear that evenings might be the best time to catch players when they're feeling kind, offering insight into how timing can influence sentiment.



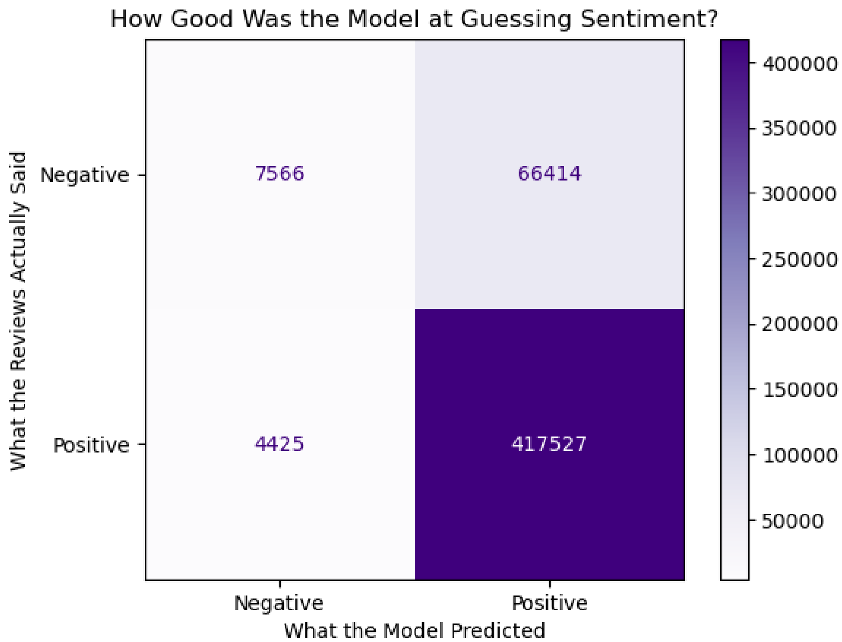
I also looked at whether players kept playing after leaving a review and how that impacts sentiment. Players who stopped playing were more likely to leave negative reviews, while those who kept playing overwhelmingly left positive ones. This makes sense—players who are still engaged are probably enjoying the game, while those who stop might already feel disconnected or let down. It's a reminder that keeping players invested doesn't just drive engagement but shapes how they feel about the game.



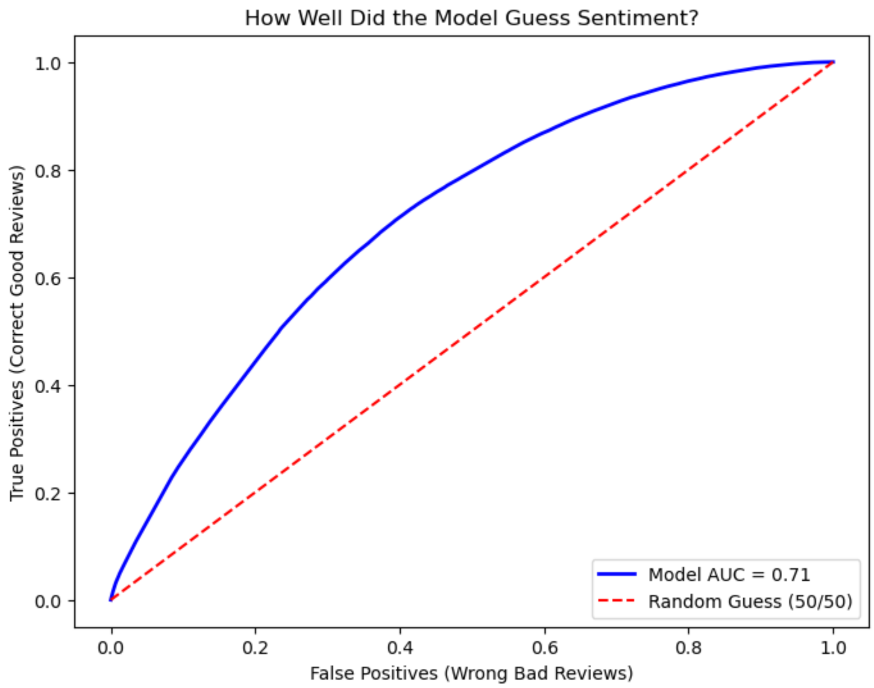
Finally, I made a heatmap to understand how the features I engineered relate to each other and to sentiment. One clear link was between author_playtime_forever and author_playtime_at_review (0.78), showing that playtime metrics capture related but slightly different behaviors. Interestingly, author_playtime_last_two_weeks had only a moderate correlation (0.33) with total playtime, highlighting the difference between recent activity and long-term engagement. Recency_bias stood out as independent, offering unique insights about timing, while author_num_games_owned had a small connection (0.31) to author_num_reviews, hinting that more active players tend to leave more feedback. None of the features were strongly correlated with voted_up, which I believe reinforces the idea that sentiment relies on a mix of influences rather than a single dominating factor.



After exploring the relationships in the data, I turned to the decision tree model itself to figure out which features were most important for predicting sentiment. Unsurprisingly, author_playtime_forever came out on top—total playtime is a strong reflection of how much someone enjoys a game. Right behind it was played_after_review, showing how ongoing engagement plays a key role in shaping sentiment. Features like time_of_day_indexed and recency_bias added to the picture, reinforcing the idea that when players leave reviews and how recent their experiences are both matter. Even smaller contributors, like author_num_reviews and author_num_games_owned, offered subtle insights into patterns of player behavior. Altogether, these results confirm what the visualizations hinted at: sentiment is shaped by a mix of engagement and timing, and no single feature tells the whole story.



To evaluate how well the model captured sentiment, I started with a confusion matrix. This chart lays out how often the model guessed review sentiment correctly versus how often it got it wrong. The results show that while the model did a solid job predicting positives, it struggled a bit with negatives, misclassifying more than 60,000 of them. Still, the number of accurate predictions, especially for positives, is reassuring and shows that the model gets the overall sentiment correct more often than not.



Finally, I created a ROC curve to get a better sense of the model's ability to separate positive and negative sentiments. The curve showed an AUC of 0.71, which means the model does better than random guessing but still has room to grow. It seems the model is pretty good at picking up on patterns, but some of the subtler differences in sentiment might still be slipping through the cracks. The AUC gives us a solid overall view, but it also shows that the model definitely needs a bit of improvement.

Final Thoughts

In wrapping up my project, I think it's safe to say it has been a really cool dive into what drives player reviews on Steam. Using a decision tree model, I set out to predict whether a review would be positive or negative based on player behavior, engagement, and timing. Along the way, I learned a lot—not just about the dataset but about how sentiment can be shaped by so many factors.

One of the biggest takeaways was that engagement metrics, like total playtime (author_playtime_forever) and whether a player kept playing after their review (played_after_review), are major influences on sentiment. It makes sense—if you're spending hours playing a game and still diving back in after leaving a review, chances are you're enjoying yourself. Timing also turned out to play a role, with features like time_of_day and recency_bias revealing subtle patterns. Players tend to leave reviews that are more positive in the evenings, and reviews written closer to when someone last played a game often reflect stronger feelings, whether good or bad. These findings reinforced that player sentiment isn't random; it's tied to behavior and context.

The model itself performed decently well, with a cross-validated accuracy of 85.61%. It was pretty good at picking up on patterns, especially when identifying positive reviews. The confusion matrix showed that while the model nailed a lot of positives, it struggled with negatives, misclassifying a decent chunk of them as positives. The ROC curve backed this up, with an AUC of 0.71 showing the model is solid but has room to grow. This tells me that while the model does a good job overall, it could be improved to catch those subtler patterns that might lead to negative reviews being missed.

Beyond the numbers, what's been really interesting is seeing how player behavior translates into sentiment trends. The visualizations made it clear that things like when players review and how engaged they are can tell a deeper story about what drives positive or negative feedback. It's also made me reflect on my own reviews—do I follow these trends? Would my playtime or the time of day I write reviews influence my sentiment? I'm going to enjoy reflecting on this project the next time I finish a game and seeing how my personal feelings match the models findings.

Overall, this Steam project was a mix of challenges and fun discoveries, and has been insanely informative over all. From wrangling the massive Steam dataset, which was very intimidating initially, to engineering features and testing a decision tree model, I've learned a lot about how data engineering can be used to unpack something as subjective as player sentiment. There's definitely room to improve my model, but the results I managed to get were still very exciting.

Appendix A

```
kaggle datasets download -d kieranpoc/steam-reviews

unzip steam-reviews.zip

gcloud storage buckets create gs://gamasteamreviews --
project=gamasteam \ --default-storage-class=STANDARD --
location=us-central1 --uniform-bucket-level-access

gcloud storage cp all_reviews/all_reviews.csv
gs://gamasteamreviews/landing/
```

Appendix B

```
import matplotlib as plt

import seaborn as sns

from pyspark.sql import functions as F

csv = "gs://gamasteamreviews/landing/all_reviews.csv"

df = spark.read.csv(csv, header=True, inferSchema=True,
multiLine=True, escape='"')



df.write.mode("overwrite").parquet("gs://gamasteamreviews/landin
g/all_reviews.parquet")

df =
spark.read.parquet("gs://gamasteamreviews/landing/all_reviews.pa
rquet")

#I converted it to parquet to try and alleviate some performance
issues

#the record counts

df.cache

df.count()
```

```
#the columns and data types

df.printSchema()

#handling the null values in the data

null_counts = df.select([F.count(F.when(F.col(c).isNull(),
c)).alias(c) for c in df.columns])

null_counts_pandas = null_counts.toPandas().transpose()

null_counts_pandas.columns = ["Null Count"]

null_counts_pandas.style

stats = df.select(

    "author_num_games_owned",

    "author_num_reviews",

    "author_playtime_forever",

    "author_playtime_last_two_weeks",

    "author_playtime_at_review",

    "author_last_played",

    "voted_up",
```

```
    "votes_up",

    "votes_funny",

    "weighted_vote_score"

).summary("count", "min", "max", "mean","stddev")


stats_pandas = stats.toPandas()


stats_pandas.style


#the stats for the dates


date_stats = df.select(

    F.from_unixtime("timestamp_created").alias("created_date"),

    F.from_unixtime("timestamp_updated").alias("updated_date")

    ).select(

    F.min("created_date").alias("min_created_date"),

    F.max("created_date").alias("max_created_date"),
```

```
    F.min("updated_date").alias("min_updated_date"),

    F.max("updated_date").alias("max_updated_date")

)



date_summary_pandas = date_stats.toPandas()

date_summary_pandas.style

#review statistics



df = df.withColumn("review_word_count",
F.size(F.split(F.col("review"), " ")))

review_stats = df.agg(

    F.min("review_word_count").alias("min_word_count"),

    F.max("review_word_count").alias("max_word_count"),

    F.avg("review_word_count").alias("avg_word_count")

)

review_stats_pandas = review_stats.toPandas()

review_stats_pandas.style

#review count by language
```

```
review_count_by_language =
df.groupBy("language").count().orderBy("count",
ascending=False).limit(5).toPandas()


sns.barplot(data=review_count_by_language, x="language",
y="count", palette="viridis")
plt.title("Review Count by Language (Top 5 Languages)")
plt.xlabel("Language")
plt.ylabel("Review Count")
plt.show()


#number of reviews for playtimes
df_sample = df.sample(0.1).select(
    (F.col("author_playtime_forever") /
60).alias("author_playtime_hours"),
    "author_num_reviews"
).toPandas()
```

```
sns.scatterplot(data=df_sample, x="author_playtime_hours",
y="author_num_reviews", alpha=0.6)

plt.title("Playtime (Hours) vs. Number of Reviews")

plt.xlabel("Playtime (Hours)")

plt.ylabel("Number of Reviews")

plt.show()
```

Appendix C

```
from pyspark.sql.types import StructType, StructField, IntegerType,
FloatType, StringType, BooleanType, LongType
from pyspark.sql import functions as F

csv = "gs://gamasteamreviews/landing/all_reviews.csv"

df = spark.read.csv(
    csv,
    header=True,
    inferSchema=True,
    multiLine=True,
    escape='"'
)
df.write.mode("overwrite").parquet("gs://gamasteamreviews/landing/all_r
eviews_parquet")
parquet_path = "gs://gamasteamreviews/landing/all_reviews_parquet"
df = spark.read.parquet(parquet_path)

columns_to_keep = [
```

```
    "author_num_games_owned",
    "author_num_reviews",
    "author_playtime_forever",
    "author_playtime_last_two_weeks",
    "author_playtime_at_review",
    "author_last_played",
    "language",
    "voted_up",
    "steam_purchase",
    "received_for_free",
    "timestamp_created",
    "timestamp_updated"
]

df = df.select(*columns_to_keep)

for col in ["author_num_games_owned", "author_num_reviews",
"author_playtime_forever",
            "author_playtime_last_two_weeks",
"author_playtime_at_review", "author_last_played"]:
    median_value = df.approxQuantile(col, [0.5], 0.05)[0]
    df = df.fillna({col: median_value})
```

```
df = df.fillna({
    "voted_up": False,
    "steam_purchase": False,
    "received_for_free": False,
})

df = df.fillna({"language": "unknown"})
cleaned_parquet_path =
"gs://gamasteamreviews/cleaned/all_reviews_cleaned.parquet"
df.write.mode("overwrite").parquet(cleaned_parquet_path)
```

Appendix D

```
from pyspark.sql import functions as F
from pyspark.sql.functions import when, col, from_unixtime, hour
from pyspark.ml.feature import StringIndexer, VectorAssembler
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
from pyspark.ml import Pipeline


df =
spark.read.parquet("gs://gamasteamreviews/cleaned/all_reviews_cleaned.p
arquet")
#with this dataset containing unix data for when the reviews are
written, it's possible that the time of day affects review sentiment.
#perhaps gamers are more likely to leave positive reviews in the
afternoon, or night.
df = df.withColumn(
    "time_of_day",
    when((hour(from_unixtime(col("timestamp_created"))) >= 6) &
(hour(from_unixtime(col("timestamp_created"))) < 12), "morning")
    .when((hour(from_unixtime(col("timestamp_created"))) >= 12) &
(hour(from_unixtime(col("timestamp_created"))) < 18), "afternoon")
```

```
    .when((hour(from_unixtime(col("timestamp_created"))) >= 18) &
(hour(from_unixtime(col("timestamp_created"))) < 24), "evening")
    .otherwise("night"))

    #this feature shows us the amount of time that has passed between
when the reviewer had played the game, and when they reviewed it
#the idea is the shorter the gap between the time last played and time
of review, the stronger the reviewer will feel about the game due to
recency bias

df = df.withColumn(
    "recency_bias",
    (col("timestamp_created") - col("author_last_played")) / (24 * 60 *
60)
)

#this is a feature piggybacking off the previous one, it checks to see
if the author has kept playing the game after their review.

df = df.withColumn(
    "played_after_review",
```

```
    F.when(col("author_last_played") > col("timestamp_created"),
1).otherwise(0)
)
#filling any missing data in our newly created numeric columns

for col_name in [
    "recency_bias",
    "played_after_review"
]:
    median_value = df.approxQuantile(col_name, [0.5], 0.05)[0]
    df = df.fillna({col_name: median_value})
#using the string indexer on our categorical values. using
handleinvalid keep to deal with missing values

language_indexer = StringIndexer(inputCol="language",
outputCol="language_indexed", handleInvalid="keep")
time_of_day_indexer = StringIndexer(inputCol="time_of_day",
outputCol="time_of_day_indexed", handleInvalid="keep")
#after using the string indexer and dealing with any missing values, we
assemble our features into a vector

feature_columns = [
```

```
    "author_num_games_owned",
    "author_num_reviews",
    "author_playtime_forever",
    "author_playtime_last_two_weeks",
    "author_playtime_at_review",
    "author_last_played",
    "recency_bias",
    "played_after_review",
    "language_indexed",
    "time_of_day_indexed"
]
assembler = VectorAssembler(inputCols=feature_columns,
outputCol="features")
#creating a pipeline

pipeline = Pipeline(stages=[language_indexer, time_of_day_indexer,
assembler])
df_transformed = pipeline.fit(df).transform(df)
#creating our data isnto training and test sets

train_data, test_data = df_transformed.randomSplit([0.7, 0.3], seed=42)
```

```
#setting up for and executing k fold cross evaluation to find the
parameters that will give us the best performance

dt_classifier = DecisionTreeClassifier(labelCol="voted_up",
featuresCol="features", maxDepth=15)

params = (
    ParamGridBuilder()
    .addGrid(dt_classifier.maxDepth, [5, 10, 15])
    .addGrid(dt_classifier.maxBins, [32, 64])
    .build()
)
evaluator = MulticlassClassificationEvaluator(labelCol="voted_up",
predictionCol="prediction", metricName="accuracy")

cross_val = CrossValidator(
    estimator=dt_classifier,
    estimatorParamMaps=params,
    evaluator=evaluator,
    numFolds=5
)
```

```
cv_model = cross_val.fit(train_data)
predictions = cv_model.bestModel.transform(test_data)
accuracy = evaluator.evaluate(predictions)
print(f"Cross-validated Accuracy: {accuracy}")
precision = MulticlassClassificationEvaluator(
    labelCol="voted_up",
    predictionCol="prediction",
    metricName="weightedPrecision"
).evaluate(predictions)
print(f"Precision: {precision}")
recall = MulticlassClassificationEvaluator(
    labelCol="voted_up",
    predictionCol="prediction",
    metricName="weightedRecall"
).evaluate(predictions)
print(f"Recall: {recall}")

f1 = MulticlassClassificationEvaluator(
    labelCol="voted_up",
    predictionCol="prediction",
    metricName="f1"
).evaluate(predictions)
```

```
print(f"F1-Score: {f1}")

trusted_path =

"gs://gamasteamreviews/Trusted/all_reviews_with_features.parquet"

df_transformed.write.mode("overwrite").parquet(trusted_path)

models_path = "gs://gamasteamreviews/Models/decision_tree_model"

cv_model.bestModel.write().overwrite().save(models_path)
```

APPENDIX E

```
from pyspark.ml.classification import DecisionTreeClassificationModel,
BinaryClassificationEvaluator
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay,
roc_curve, auc
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

trusted_path =
"gs://gamasteamreviews/Trusted/all_reviews_with_features.parquet"
df_transformed = spark.read.parquet(trusted_path)

models_path = "gs://gamasteamreviews/Models/decision_tree_model"
cv_model = DecisionTreeClassificationModel.load(models_path)
sample = df_transformed.sample(fraction=0.01, seed=42)
time_sample_pd = sample.select("time_of_day", "voted_up").toPandas()

time_sample_pd["time_of_day"] = pd.Categorical(
    time_sample_pd["time_of_day"],
    categories=["morning", "afternoon", "evening", "night"],
    ordered=True
```

```
)

sns.countplot(data=time_sample_pd, x="time_of_day", hue="voted_up",
palette="Set2")
plt.title("Sentiment Distribution by Time of Day")
plt.xlabel("Time of Day")
plt.ylabel("Number of Reviews")
plt.legend(title="Sentiment", labels=["Negative", "Positive"])
plt.show()
proportions = (
    time_sample_pd.groupby("time_of_day")["voted_up"]
    .value_counts(normalize=True)
    .rename("proportion")
    .reset_index()
)

negative_proportions = proportions[proportions["voted_up"] == 0]

ax = sns.barplot(data=negative_proportions, x="time_of_day",
y="proportion", palette="Set2")

for bar in ax.patches:
```

```
    ax.annotate(
        f"{bar.get_height():.2f}",
        (bar.get_x() + bar.get_width() / 2, bar.get_height()),
        ha="center",
        va="bottom",
        fontsize=10
    )

plt.title("Proportion of Negative Reviews by Time of Day")
plt.xlabel("Time of Day")
plt.ylabel("Proportion of Negative Reviews")
plt.ylim(0, 1)
plt.show()
heatmap_sample_pd = sample.select(
    "recency_bias",
    "author_playtime_forever",
    "author_playtime_last_two_weeks",
    "author_playtime_at_review",
    "author_num_games_owned",
    "author_num_reviews",
    "voted_up"
).toPandas()
```

```
correlation_matrix = heatmap_sample_pd.corr()

sns.heatmap(correlation_matrix, annot=True, fmt=".2f", cmap="coolwarm",
cbar=True)
plt.title("Correlation Heatmap of Engineered Features")
plt.show()
played_after_sample_pd = sample.select("played_after_review",
"voted_up").toPandas()

proportions = (
    played_after_sample_pd.groupby("played_after_review")["voted_up"]
    .value_counts(normalize=True)
    .rename("proportion")
    .reset_index()
)

ax = sns.barplot(data=proportions, x="played_after_review",
y="proportion", hue="voted_up", palette="Set2")
plt.title("Proportion of Sentiment by Played After Review")
plt.xlabel("Played After Review (0 = No, 1 = Yes)")
plt.ylabel("Proportion of Reviews")
```

```
handles, labels = ax.get_legend_handles_labels()
plt.legend(handles=handles, labels=["Negative", "Positive"],
title="Sentiment")

plt.show()
imp = cv_model.featureImportances

features = [
    "author_num_games_owned",
    "author_num_reviews",
    "author_playtime_forever",
    "author_playtime_last_two_weeks",
    "author_playtime_at_review",
    "recency_bias",
    "played_after_review",
    "language_indexed",
    "time_of_day_indexed"
]

imp_dict = {}
for i in range(len(features)):
```

```
        feature_name = features[i]
        feature_importance = imp[i]
        imp_dict[feature_name] = feature_importance

imp_list = []
for key in imp_dict.keys():
    imp_list.append((key, imp_dict[key]))

sorted_imp = []
for item in sorted(imp_list, key=lambda x: x[1], reverse=True):
    sorted_imp.append(item)

print("Feature Importances:")
for i in range(len(sorted_imp)):
    feature = sorted_imp[i][0]
    importance = round(sorted_imp[i][1], 4)
    print(feature + ": " + str(importance))

feature_names = []
feature_values = []
for item in sorted_imp:
    feature_names.append(item[0])
```

```
    feature_values.append(item[1])

imp_df = pd.DataFrame({"Feature": feature_names, "Importance":
feature_values})

sns.barplot(data=imp_df, x="Importance", y="Feature",
palette="viridis")
plt.title("Feature Importances in Decision Tree Model")
plt.xlabel("Importance")
plt.ylabel("Feature")
plt.show()

predictions = cv_model.transform(df_transformed)

sampled_preds = predictions.sample(fraction=0.01, seed=42)

sampled_preds_pd = sampled_preds.select("voted_up", "prediction",
"probability").toPandas()

true_sentiments = sampled_preds_pd["voted_up"]
guessed_sentiments = sampled_preds_pd["prediction"]
```

```
conf_matrix = confusion_matrix(true_sentiments, guessed_sentiments)

conf_matrix_display =
ConfusionMatrixDisplay(confusion_matrix=conf_matrix,
display_labels=["Negative", "Positive"])
conf_matrix_display.plot(cmap="Purples", colorbar=True)
plt.title("How Good Was the Model at Guessing Sentiment?")
plt.xlabel("What the Model Predicted")
plt.ylabel("What the Reviews Actually Said")
plt.show()

sampled_probs_pd = sampled_preds.select("voted_up",
"probability").toPandas()

true_sentiments = sampled_probs_pd["voted_up"]
positive_probs = sampled_probs_pd["probability"].apply(lambda x: x[1])

fpr, tpr, thresholds = roc_curve(true_sentiments, positive_probs)
roc_score = auc(fpr, tpr)

plt.plot(fpr, tpr, color="blue", lw=2, label=f"Model AUC =
{roc_score:.2f}")
```

```
plt.plot([0, 1], [0, 1], color="red", linestyle="--", label="Random
Guess (50/50)")
plt.title("How Well Did the Model Guess Sentiment?")
plt.xlabel("False Positives (Wrong Bad Reviews)")
plt.ylabel("True Positives (Correct Good Reviews)")
plt.legend(loc="lower right")
plt.show()
```