Exploratory Data Analysis

The data is loaded into a dataframe. It contains 12 features and 3825 entries. Most features are integers.

Context

There are three features that are missing about 48% of their features; namely, price, threadvalue, CPUvalue. Power performance and TDP are missing about 18% of their features. These values have just been garnered by finding the null values in the dataframe. In reality, many socket types and some category data (the feature) is missing but just has the entry “unknown”. So, the first lot of preprocessing completed on this data is changing “unknown” and “Unknown” to a null value and representing these percentages as a graph.

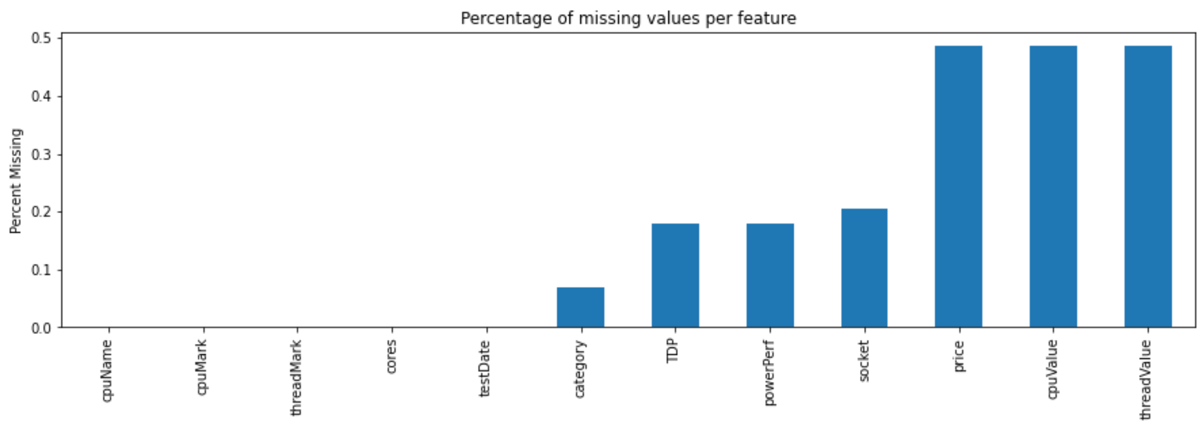

Figure 1: Percentage of data missing.

No features were removed. Instead, all missing values were imputed from the data. Instead of filling in all price columns with the mean price, two approaches were trialled. The first was splitting the dataframe into categories with the major CPU lines: 'AMD Ryzen', 'AMD EPYC', 'Intel Xeon', 'Intel Core', ‘Other’, etc. However, it was difficult to categorise all data by hand. Approximately 40% of the data got lodged into ‘Other’ as they did not fit into the specific line of ‘Intel’. Instead, CPU names were simply categorised by companies into the major categories: AMD, Intel, Qualcomm, Samsung. This only resulted in 9% of the data categorised as ‘other’ which was cpus without a company name, and other smaller companies such as snapdragon etc.

There are tradeoffs and limtiations for both approaches. The first approach results in half of the data being imputed by the global mean (the mean across cpus that were not categorised). The second approach results in losing the nuance of prices, performance between different company lines e.g. AMD Ryzen and AMD Epyc. However, it does result in less data being imputed by some uncategorised average. So, the second approach was chosen for imputation. In hindsight, it would’ve been easier to split the cpuName column by the first two strings and use that for imputation.

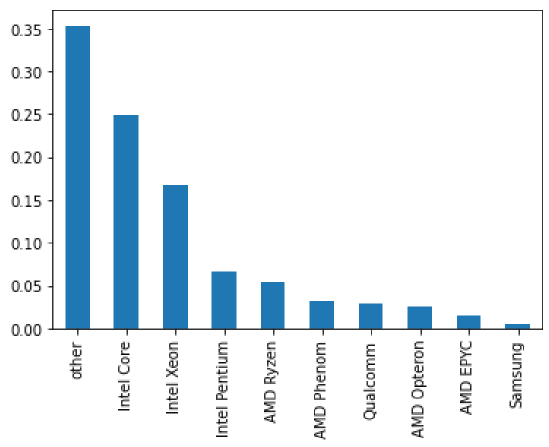

Fig 2: Approach 1 for categories for imputation

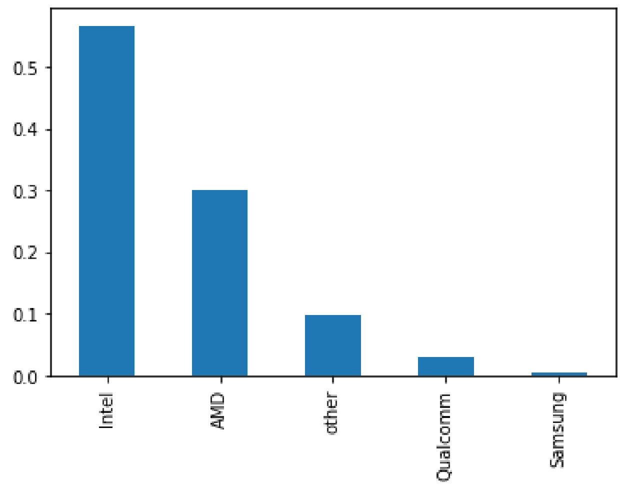

Fig 3: Approach 2.

The mean of each of these respective categories was used to impute the continuous data. So, for example, if a CPU started with the name “Intel”, all rows with the same name were sorted into their own data frame, and then all price values in that smaller data frame were imputed using the mean for the category “Intel”. This code is shown below.

```
values = ['AMD', 'Intel','Qualcomm', 'Samsung']

conditions = list(map(df['cpuName'].str.contains, values))

df['CpuCats'] = np.select(conditions, values, 'other')
categories = []
for i in list(set(df['CpuCats'])):
    df_cat = df[df['CpuCats']== i]
    df_cat['price'].fillna(df_cat['price'].mean(),inplace = True)
    df_cat['cpuValue'].fillna(df_cat['cpuValue'].mean(),inplace = True)
    df_cat['TDP'].fillna(df_cat['TDP'].mean(),inplace = True)
    df_cat['threadValue'].fillna(df_cat['threadValue'].mean(),inplace =
True)
    df_cat['powerPerf'].fillna(df_cat['powerPerf'].mean(),inplace =
True)



    categories.append(df_cat)
    final_df = pd.concat(categories)
```

Some numerical data was still missing after this categorical imputation. This was probably because some categories did not have means. So, these were filled with the global mean. Categorical data was not imputed, but instead just uses “unknown”

Correlation Matrix

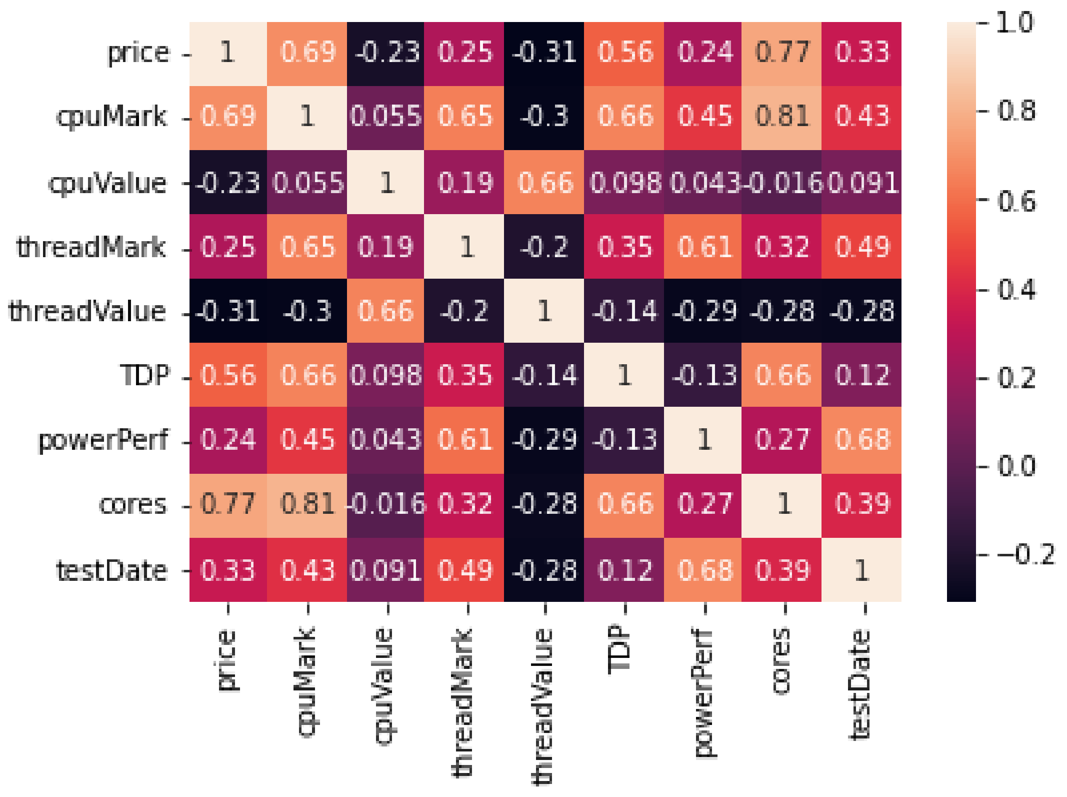

Fig4: Correlation Matrix.

Some features appear to be highly correlated, such as threadvalue and TDP, as well as CPUMark and CPUvalue. This information will be used later when improving naïve bayes classifier. Cores and Cpu mark are very highly correlated.

Density plot of features

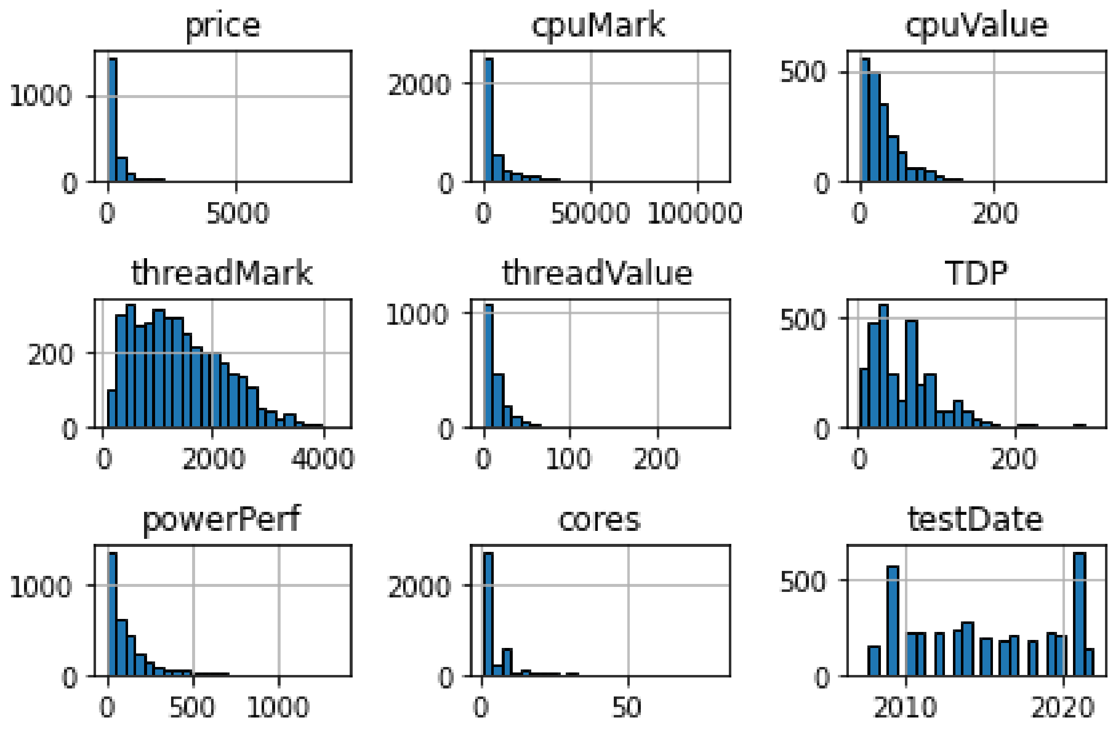

Fig 5: Density plot.

There are some common densities in this data. Cores, Price, CpuMark, and threadvalue all appear to have the same spread. Threadvalue and Price were 48% missing and imputed using the same way, so we could be seeing the impact of imputation here. However, the fact that CPU mark which didn't have any imputation procedure done to it has the same spread suggests this is a feature of the data, not its preprocessing.

The Problem

This is going to be a classification problem using the 10 features to predict the type of category the cpu is. The issue with this currently is that some categories have multiple values in the entry. I've spent enough time on data processing so these will be their own labels e.g. "Laptop" and "Laptop/Embedded" are different labels because it captures the nuance of the cpu and reducing them to one is reductive.

Three types of classifiers will be used for this problem: naïve bayes, SVMs, and MLPs. Hyperparameters will be tuned and other ways will be explored on how to improve them.

The testdate feature has been removed because the year of testing does not have anything to do with the category of the CPU; if the date was when the CPU had been released, then this could've provided some information to the task. However, it has been removed because it will not provide much useful information.

The data has been split into training and testing data via the train_test_split method in sklearn. The testing data will not be used at all for cross-validation; only to get an accuracy score at the end of each classifer. The train and test split was completed using a random_state parameter set to 0. This was set so the same split would be given each time. I'll explain why this was necessary in the relevant section.

Dimensionality Reduction

PCA was completed using the sklearn library. It resulted in the following scree graph. Evidently, the first principal component explains 92% of the variance of the data. It is clear that 2 components can be used for PCA.

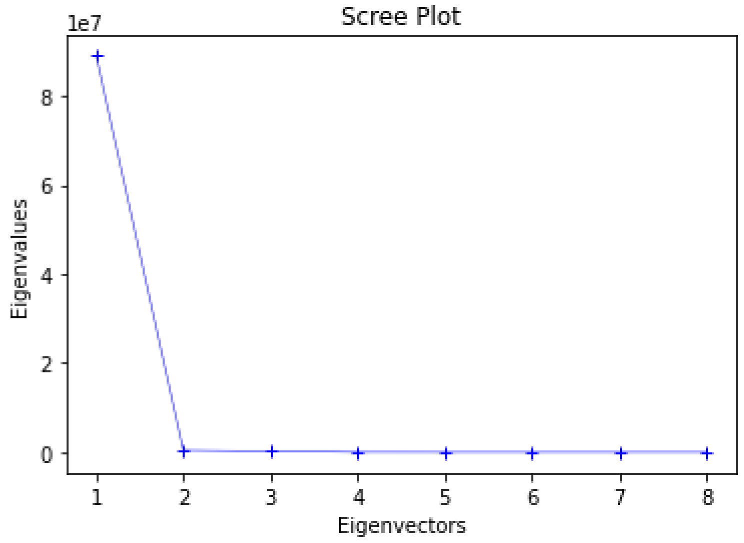

Figure 6: Scree Plot.

Only one PCA was used to transform to the train and test data.

```

pca = PCA(2)
pca.fit(x_train)

x_train_pca = pca.transform(x_train)
x_test_pca = pca.transform(x_test)
```

Naive Bayes

GaussianNB from sklearn was used to run a Naïve Bayes classifier. The sample code to run one classifier is given below. The cross_val_score method has the parameter cv=10; this means a stratified 10-fold cross validation strategy was used on the training set. I later print out the mean performance of this cross validation and use that in the table below.

```
base_classifier = GaussianNB()
base_classifier.fit(x_train, y_train)
cv_score = cross_val_score(base_classifier, x_train, y_train, cv=10)
y_pred = base_classifier.predict(x_test)
print(cv_score.mean())
print('%s classifer accuracy: %f' % ("Naive
Bayes",accuracy_score(y_pred, y_test)*100))
```

Firstly, I ran two classifiers, one using PCA and one without. The base data performed much better than PCA. This may be because the PCA results for this classification problem were quite odd; most of the variance in the data was packed into one principal component. PCA is usually expected to improve Naïve bayes because it attempts to transform data into a new uncorrelated dataset.

Two other methods were used to attempt to improve the naïve bayes classification score. These are: removing correlated features using the correlation matrix in figure 4, and using KDE. Naïve bayes is called that because it works on the naïve assumption that there is conditional independence between every pair of features given the class variable. This results in highly correlated features being effectively counted twice, so that's why I remove one of the correlated features. I removed threadvalue because it is highly correlated with TDP, and CPUMark because it is highly correlated with cores.

	Mean cross validation score	Test set accuracy
Base data	57.90%	57.77%
Removed correlated features	62.32%	63.52%
Using KDE Bandwidth = 100, kernel=gaussian	61.01%	61.96%
Using PCA	37.54%	38.43%
KDE + removed highly correlated features. Bandwidth = 30, kernel = gaussian	65.75%	64.70%

Finally, I attempted to improve naïve bayes estimator via kernel density estimation. In naïve bayes, it is assumed that each underlying prior distribution is a gaussian distribution. Instead, I tried to estimate the actual underlying distribution via kernel density estimation. I adapted code from an online source and got approval to use it. I understand this is a deviation from the course material, but its just an extension I wanted to present in my report.
https://edstem.org/au/courses/7692/discussion/899501?answer=2030517

I used a variety of kernel functions, and most presented worse or similar results to the guassian kernel. I then used a for loop to tune the kernel bandwidth which controls the size of the kernel at each point. Unfortunately, as from the table, this method did not show any improvement in the results. It improved the method, however, was still slightly below the results from removing correlated features. This anti-climactic result tells me one thing about the data given the behaviour of the classifier: removing highly correlated features was much more important than density estimation. Instead, it appears that the naïve gaussian assumption was largely ignored due to the dimensionality of the data. This is why naïve bayes classifiers tend to perform well in text classification tasks; as the dimensionality of data increases, the distribution of two features tends to matter less and less.

I tried one final method to improve the naïve bayes classifier, which was to use KDE estimation on the dataset with the highly correlated features removed. This combination performed best. Interestingly, the kernel that performed best on this data was lower, at 30. A lower kernel bandwidth changes the shape of the kernel; it means only points very close to the current position are given any weight which results in a tighter density function overall.

SVM

The second classifier I used was SVM. Unlike Naïve Bayes which generates a probability to determine labels for new points, SVM is an example of discriminative classification. Instead of modelling each class, instead SVM attempts to find a manifold that divides categories from each other.

There are two main parameters in SVM classification: kernel function and C value. The kernel functions in SVM include linear, polynomial kernel and rbfs kernel. SVM with a linear kernel is a linear classifier; using other kernel functions allows it to fit non-linear relationships. C controls the hardness of the margin between classes. A large value of C makes the margin hard, and points cannot lie in it. A smaller C results in a softer margin, and points are able to lie in it.

To tune this parameter, I used sklearn’s gridsearch on the x train, y_train data. Its been 9 minutes and grid search is still running. I analyse the trends in these results using some graphs below.

```
from sklearn.svm import SVC

svc = SVC()

kernels = ['linear', 'poly', 'rbf']
c = np.linspace(1,100, 10)


grid = GridSearchCV(svc, param_grid={'kernel':kernels, 'C':c})
grid.fit(x_train, y_train)

print(grid.best_params_)
print(grid.cv_results_['params'])
print(grid.cv_results_['mean_test_score'])
print(grid.cv_results_['std_test_score'])

results = grid.cv_results_
```

SVM is running quite slow. I’ve waited five minutes for one single call to SVM fit()– not even the grid search below. This is quite heartbreaking since I spent a lot of time researching SVM; I think I should’ve played around with more highly dimensional datasets to see how long SVC from sklearn would take to run on those. I’ll skip SVM for now.

MLP

The final classifier I want to run is a MLP. I chose MLP as the final classifier because it has a range of interesting hyperparmeters I wanted to tune to improve the behaviour of the base classifier. I was initially going to use sklearn’s gridsearch to compare combinations of different parameters. However, a grid search with 6 possible combinations took longer than 10 minutes to run. Not having this time, instead, I ran a few for loops to run these combinations. I used cv=10 for cross val score to complete cross validation across the test set. I used the MLPClassifier from sklearn to perform this task.

Firstly, I tuned the activation function used for the MLP. This activation function computes how the input values of a layer is presented to the output value. The standard activation function is the linear function. However, the derivative of this activation function is constant, so it is not possible to use backpropogation to train the model and understand which weights can provide a better prediction. Out of the logistic and relu functions, the best performing activation function was the logistic function. This normalises ouput between 0 and 1, effectively outputting the probability of the input value. It is useful for classification-aimed neural networks. It can sometime result in vanishing gradients where if the result is 0, then there is no update in weights. The cross validation score for the logistic function was 69%, the best performing rate so far.

The second parameter to tune was the learning rate. I tried the range of 0.001 to 0.1. Here, Lower learning rates performed best regardless of which activation function was used. This resulted in improved accuracy, however, led to slower convergence overall since the stepsize was smaller (learning rate is the step size). Since 0.001 was the best performing learning rate, I tried to go even lower to 0.0001 and this resulted in cross-validation accuracy of:

However, this smaller range of learning rates (from 0.0001) onwards resulted in the MLP not being able to converge in the default number of iterations of 200. I increased this to 1000, and the MLP was still not able to converge. This means the step size was too small. However, the cross-validation score was 71%!

As a result, the final hyperparameter I tuned was momentum. Momentum can help accelerate gradient vectors in the right direction and lead to faster converging. Integrating momentum can stabilise update directions and allow the algorithm to escape from local maxima. Momentum method is a technique for accelerating gradient descent algorithms by accumulating a velocity vector in gradient direction of loss function across iterations. Memorisation of previous gradients allows algorithm to quickly overcome bumps in the loss surface. It works by allowing an algorithm to build inertia towards a specific direction.

Running the following MLP classifier, with momentum = 0.99 instead of default 0.9, learning rate = 0.001, and activation of logisitic function resulted in cross validation accuracy of 69.05%. Further more, the classifier was actually able to converge in the default 200 iterations due to the increased momentum parameter. This was also the first classifier that resulted in equal cross validation and test set accuracy.

	Mean cross validation score	Test set accuracy
Base	56%	53%
Final	69%	69%

A snippet of MLP code.

```
from sklearn.model_selection import cross_val_score
clf = MLPClassifier(activation='logistic', learning_rate_init=0.001,
momentum=0.99)
cv_score = cross_val_score(clf, x_train, y_train, cv=10)
print(cv_score.mean())
clf.fit(x_train, y_train)
y_pred = clf.predict(x_test)
print("Accuracy of Base MLP Classifier
{:.0%}".format(accuracy_score(y_pred, y_test)))
```

Conclusions

I was able to take an incomplete dataset and perform some considered imputations on the data. I ran two distinct types of classifiers and attempted to improve classifier performance through a range of methods.